Definitions by Rewriting in the Calculus of Constructions

Frédéric Blanqui

LRI, bât. 490, Université Paris-Sud, 91405 Orsay Cedex, France

tel: +33 (0) 1 69 15 42 35 fax: +33 (0) 1 69 15 65 86

blanqui@lri.fr

Abstract : *The main novelty of this paper is to consider an extension of the Calculus of Constructions where predicates can be defined with a general form of rewrite rules.*

We prove the strong normalization of the reduction relation generated by the β-rule and the user-defined rules under some general syntactic conditions including confluence.

As examples, we show that two important systems satisfy these conditions : a sub-system of the Calculus of Inductive Constructions which is the basis of the proof assistant Coq, and the Natural Deduction Modulo a large class of equational theories.

1 Introduction

This work aims at defining an expressive language allowing to specify and prove mathematical properties in which functions and predicates can be defined by rewrite rules, hence enabling the automatic proof of equational problems.

The Calculus of Constructions. The quest for such a language started with Girard's system F [19] on one hand and De Bruijn's Automath project [18] on the other hand. Later, Coquand and Huet combined both calculi into the Calculus of Constructions (CC) [10]. As in system F, in CC, data structures are defined by using an impredicative encoding which is difficult to use in practice. Following Martin-Löf's theory of types [24], Coquand and Paulin-Mohring defined an extension of CC with inductive types and their associated induction principles as first-class objects : the Calculus of Inductive Constructions (CIC) [26] which is the basis of the proof-assistant Coq [17].

Reasoning Modulo. Defining functions or predicates by recursion is not always convenient. Moreover, with such definitions, equational reasoning is uneasy and leads to very large proof terms. Yet, for decidable theories, equational proofs need not to be kept in proof terms. This idea that proving is not only reasoning (undecidable) but also computing (decidable) has been recently formalized in a general way by Dowek, Hardin and Kirchner with the Natural Deduction Modulo (NDM) for first-order logic [12].

Object-level rewriting. In CC, the first extension by a general notion of rewriting is the λR-cube of Barbanera, Fernández and Geuvers [1]. Their work extends the works of Breazu-Tannen and Gallier [8] and Jouannaud and Okada [21] on the combination of typed λ-calculi with rewriting. The notion of rewriting considered in [21, 1] is not restricted to first-order rewriting, but also includes higher-order rewriting following Jouannaud and Okada's General Schema [21], a generalization of the primitive recursive definition schema. This schema has been reformulated and enhanced so as to deal with definitions on strictly-positive inductive types [5] and with higher-order pattern-matching [3].

Predicate-level rewriting. The notion of rewriting considered in [1] is restricted to the object-level while, in CIC or NDM, it is possible to define predicates by recursion or by rewriting respectively. Recursion at the predicate-level is called "strong elimination" in [26] and has been shown consistent by Werner [31].

Our contributions. The main contribution of our work is a strong normalization result for the Calculus of Constructions extended with, at the predicate-level, user-defined rewrite rules satisfying some general admissibility conditions. As examples, we show that these conditions are satisfied by a sub-system of CIC with strong elimination [26] and the Natural Deduction Modulo [13] a large class of equational theories.

So, our work can be used as a foundation for an extension of a proof assistant like Coq [17] where users could define functions and predicates by rewrite rules. Checking the admissibility conditions or the convert-

ibility of two expressions may require the use of external specialized tools like CiME [16] or ELAN [15].

Outline of the paper. In Section 2, we introduce the Calculus of Algebraic Constructions and our notations. In Section 3, we present our general syntactic conditions. In Section 4, we apply our result to CIC and NDM. In Section 5, we summarize the main contributions of our work and, in Section 6, we give future directions of work. Detailed proofs can be found in [4].

2 The Calculus of Algebraic Constructions (CAC)

2.1 Syntax and notations

We assume the reader familiar with the basics of rewriting [11] and typed λ-calculus [2].

Sorts and symbols. Throughout the paper, we let $\mathcal{S} = \{\star, \Box\}$ be the set of *sorts* where $\star$ denotes the impredicative universe of propositions and $\Box$ a predicative universe containing $\star$. We also assume given a family $\mathcal{F} = (\mathcal{F}^s_n)^{s\in\mathcal{S}}_{n\geq 0}$ of sets of *symbols* and a family $\mathcal{X} = (\mathcal{X}^s)^{s\in\mathcal{S}}$ of infinite sets of *variables*. A symbol $f \in \mathcal{F}^s_n$ is said to be of *arity* $\alpha_f = n$ and sort s. $\mathcal{F}^s$, $\mathcal{F}_n$, $\mathcal{F}$ and $\mathcal{X}$ respectively denote the set of symbols of sort s, the set of symbols of arity n, the set of all symbols and the set of all variables.

Terms. The *terms* of the corresponding CAC are given by the following syntax :

$$t ::= s \mid x \mid f(\vec{t}) \mid (x:t)t \mid [x:t]t \mid tt$$

where $s \in \mathcal{S}$, $x \in \mathcal{X}$ and f is applied to a vector $\vec{t}$ of n terms if $f \in \mathcal{F}_n$. $[x:U]t$ is the abstraction and $(x:U)V$ is the product. A term is *algebraic* if it is a variable or of the form $f(\vec{t})$ with each t_i algebraic.

Notations. As usual, we consider terms up to α-conversion. We denote by $FV(t)$ the set of free variables of t, by $FV^s(t)$ the set $FV(t) \cap \mathcal{X}^s$, by $t\{x \mapsto u\}$ the term obtained by substituting in t every free occurrence of x by u, by $dom(\theta)$ the domain of the substitution θ, by $dom^s(\theta)$ the set $dom(\theta) \cap \mathcal{X}^s$, by $Pos(t)$ the set of positions in t (words on the alphabet of positive integers), by $t|_p$ the subterm of t at position p, by $t[u]_p$ the term obtained by replacing $t|_p$ by u in t, and by $Pos(f,t)$ and $Pos(x,t)$ the sets of positions in t where f occurs and x freely occurs respectively. As usual, we write $T \to U$ for a product $(x:T)U$ where $x \notin FV(U)$.

Rewriting. We assume given a set $\mathcal{R}$ of *rewrite rules* defining the symbols in $\mathcal{F}$. The rules we consider are pairs $l \to r$ made of two terms l and r such that l is an algebraic term of the form $f(\vec{l})$ and $FV(r) \subseteq FV(l)$. They induce a rewrite relation $\to_{\mathcal{R}}$ on terms defined by $t \to_{\mathcal{R}} t'$ iff there are $p \in Pos(t)$, $l \to r \in \mathcal{R}$ and a substitution σ such that $t|_p = l\sigma$ and $t' = t[r\sigma]_p$ (matching is first-order). So, $\mathcal{R}$ can be seen as a particular case of Combinatory Reduction System (CRS) [23] (translate $[x:T]u$ into $\Lambda(T,[x]u)$ and $(x:T)U$ into $\Pi(T,[x]U)$) for which higher-order pattern-matching is not necessary.

Reduction. The *reduction relation* of the calculus is $\to = \to_{\mathcal{R}} \cup \to_\beta$ where $\to_\beta$ is defined as usual by $[x:T]u\,t \to_\beta u\{x \mapsto t\}$. We denote by $\to^*$ its reflexive and transitive closure, by $\leftrightarrow^*$ its symmetric, reflexive and transitive closure, and by $t \downarrow^* u$ the fact that t and u have a common reduct.

2.2 Typing

Types of symbols. We assume given a function τ which, to each symbol f, associates a term τ_f, called its *type*, of the form $(\vec{x} : \vec{T})U$ with $|\vec{x}| = \alpha_f$. In contrast with our own previous work [5] or the work of Barbanera, Fernández and Geuvers [1], symbols can have polymorphic as well as dependent types, as it is the case in CIC.

Typing. An *environment* Γ is an ordered list of pairs $x_i:T_i$ saying that x_i is of type T_i. The *typing relation* of the calculus, $\vdash$, is defined by the rules of Figure 1 (where $s, s' \in \mathcal{S}$).

An environment is *valid* if there is a term typable in it. The condition $\Gamma \vdash v : V$ in the (symb) rule insures that Γ is valid in the case where $n = 0$.

Substitutions. Given two valid environments Γ and Δ, a substitution θ is a *well-typed substitution* from Γ to Δ, written $\theta : \Gamma \to \Delta$, if, for all $x \in dom(\Gamma)$, $\Delta \vdash x\theta : x\Gamma\theta$, where $x\Gamma$ denotes the type associated to x in Γ. With such a substitution, if $\Gamma \vdash t : T$ then $\Delta \vdash t\theta : T\theta$.

Logical consistency. As usual, the logical consistency of such a system is proved in three steps.

First, we must make sure that the reduction relation is correct w.r.t. the typing relation : if $\Gamma \vdash t : T$ and $t \to t'$ then $\Gamma \vdash t' : T$. This property, called *subject reduction*, is not easy to prove for extensions of CC [31, 1]. In the following subsection, we give sufficient conditions for it.

The second step is to prove that the reduction relation $\to$ is weakly or strongly normalizing, hence that every well-typed term has a normal form. Together with the confluence, this implies the decidability of the

Figure 1: Typing rules

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

$$\text{(symb)}\quad \frac{f \in \mathcal{F}^s_n,\ \tau_f = (\vec{x} : \vec{T})U,\ \gamma = \{\vec{x} \mapsto \vec{t}\} \quad \vdash \tau_f : s \quad \Gamma \vdash v : V \quad \forall i,\ \Gamma \vdash t_i : T_i\gamma}{\Gamma \vdash f(\vec{t}) : U\gamma}$$

$$\text{(var)}\quad \frac{\Gamma \vdash T : s \quad x \in \mathcal{X}^s \setminus dom(\Gamma)}{\Gamma, x : T \vdash x : T}$$

$$\text{(weak)}\quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s \quad x \in \mathcal{X}^s \setminus dom(\Gamma)}{\Gamma, x : U \vdash t : T}$$

$$\text{(prod)}\quad \frac{\Gamma \vdash T : s \quad \Gamma, x : T \vdash U : s'}{\Gamma \vdash (x : T)U : s'}$$

$$\text{(abs)}\quad \frac{\Gamma, x : T \vdash u : U \quad \Gamma \vdash (x : T)U : s}{\Gamma \vdash [x : T]u : (x : T)U}$$

$$\text{(app)}\quad \frac{\Gamma \vdash t : (x : U)V \quad \Gamma \vdash u : U}{\Gamma \vdash tu : V\{x \mapsto u\}}$$

$$\text{(conv)}\quad \frac{\Gamma \vdash t : T \quad T \downarrow^* T' \quad \Gamma \vdash T' : s'}{\Gamma \vdash t : T'}$$

typing relation which is essential in proof assistants. In this paper, we will study the strong normalization property.

The third step is to make sure that there is no normal proof of $\bot = (P : \star)P$ in the empty environment. Indeed, if $\bot$ is provable then any proposition P is provable. We will not address this problem here.

2.3 Subject reduction

Proving subject reduction for $\rightarrow_\beta$ requires the following property [4] :

$$(x : U)V \leftrightarrow^* (x : U')V' \ \Rightarrow\ U \leftrightarrow^* U' \ \wedge\ V \leftrightarrow^* V'$$

It is easy to see that this property is satisfied when $\rightarrow$ is confluent, an assumption which is part of our admissibility conditions described in the next section.

For $\rightarrow_\mathcal{R}$, the idea present in all previous works is to require that, for each rule $l \rightarrow r$, there is an environment Γ and a type T such that $\Gamma \vdash l : T$ and $\Gamma \vdash r : T$. However, this approach has an important drawback : in presence of dependent or polymorphic types, it leads to non-left-linear rules.

For example, consider the type $list : \star \rightarrow \star$ of polymorphic lists built from $nil : (A : \star)list(A)$ and $cons : (A : \star)A \rightarrow list(A) \rightarrow list(A)$, and the concatenation function $app : (A : \star)list(A) \rightarrow list(A) \rightarrow list(A)$. To fulfill the previous condition, we must define app as follows :

$$\begin{array}{rcl} app(A, nil(A), \ell) & \rightarrow & \ell \\ app(A, cons(A, x, \ell), \ell') & \rightarrow & cons(A, x, app(A, \ell, \ell')) \end{array}$$

This has two important consequences. The first one is that rewriting is slowed down because of numerous equality tests. The second one is that it may become much more difficult to prove the confluence of the rewrite relation and of its combination with $\rightarrow_\beta$.

We are going to see that we can take the following left-linear definition without loosing the subject reduction property :

$$\begin{array}{rcl} app(A, nil(A'), \ell) & \rightarrow & \ell \\ app(A, cons(A', x, \ell), \ell') & \rightarrow & cons(A, x, app(A, \ell, \ell')) \end{array}$$

Let $l = app(A, cons(A', x, \ell), \ell')$, $r = cons(A, x, app(A, \ell, \ell'))$, Γ be an environment and σ a substitution such that $\Gamma \vdash l\sigma : list(A\sigma)$. We must prove that $\Gamma \vdash r\sigma : list(A\sigma)$. For $\Gamma \vdash l\sigma : list(A\sigma)$, we must have a derivation like :

$$\text{(symb)}\ \frac{\Gamma \vdash A'\sigma : \star \quad \Gamma \vdash x\sigma : A'\sigma \quad \Gamma \vdash \ell\sigma : list(A'\sigma)}{\Gamma \vdash cons(A'\sigma, x\sigma, \ell\sigma) : list(A'\sigma)}$$

$$\text{(conv)}\ \frac{list(A'\sigma) \downarrow^* list(A\sigma) \quad \Gamma \vdash list(A\sigma) : \star}{\Gamma \vdash cons(A'\sigma, x\sigma, \ell\sigma) : list(A\sigma)}$$

$$\text{(symb)}\ \frac{\Gamma \vdash A\sigma : \star \quad \Gamma \vdash \ell'\sigma : list(A\sigma)}{\Gamma \vdash l\sigma : list(A\sigma)}$$

Therefore, $A'\sigma \downarrow^* A\sigma$ and we can derive $\Gamma \vdash x\sigma : A\sigma$, $\Gamma \vdash \ell\sigma : list(A\sigma)$ and :

$$\text{(symb)}\ \frac{\Gamma \vdash A\sigma : \star \quad \Gamma \vdash \ell\sigma : list(A\sigma) \quad \ell'\sigma : list(A\sigma)}{\Gamma \vdash app(A\sigma, \ell\sigma, \ell'\sigma) : list(A\sigma)}$$

$$\text{(symb)}\ \frac{\Gamma \vdash A\sigma : \star \quad \Gamma \vdash x\sigma : A\sigma}{\Gamma \vdash r\sigma : list(A\sigma)}$$

The point is that, although l is not typable, from any typable instance $l\sigma$ of l, we can deduce that $A'\sigma \downarrow^* A\sigma$. By this way, we come to the following conditions :

Definition 1 (Type-preserving rewrite rule) A rewrite rule $l \rightarrow r$ is *type-preserving* if there is an environment Γ and a substitution ρ such that, if $l = f(\vec{l})$, $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$ then :

(S1) $dom(\rho) \subseteq FV(l) \setminus dom(\Gamma)$,
(S2) $\Gamma \vdash l\rho : U\gamma\rho$,
(S3) $\Gamma \vdash r : U\gamma\rho$,
(S4) for any substitution σ, environment Δ and type T, if $\Delta \vdash l\sigma : T$ then $\sigma : \Gamma \rightarrow \Delta$,

(S5) for any substitution σ, environment Δ and type T, if $\Delta \vdash l\sigma : T$ then, for all $x \in dom(\rho)$, $x\sigma \downarrow^* x\rho\sigma$.

In our example, it suffices to take $\Gamma = A:\star, x:A, \ell: list(A), \ell':list(A)$ and $\rho = \{A' \mapsto A\}$.

One may wonder how to check these conditions. In practice, the symbols are incrementally defined. So, assume that we have a confluent and strongly normalizing CAC built over $\mathcal{F}$ and $\mathcal{R}$ and that we want to add a new symbol g. Then, given Γ and ρ, it is decidable to check (S1) to (S3) in the CAC built over $\mathcal{F} \cup \{g\}$ and $\mathcal{R}$ since this system is confluent and strongly normalizing. In [4], we give a simple condition ensuring (S4) (Γ simply needs to be well chosen). The condition (S5) is the most difficult to check and may require the confluence of $\rightarrow$.

3 Admissibility conditions

3.1 Inductive structure

Until now, we made few assumptions on symbols or rewrite rules. In particular, we have no notion of inductive type. Yet, the structure of inductive types plays a key role in strong normalization proofs [25]. On the other hand, we want rewriting to be as general as possible by allowing matching on defined symbols and equations among constructors. This is why, in the following, we introduce an extended notion of constructor and a notion of inductive structure which generalize usual definitions of inductive types [26]. Note that, in contrast with our previous work [5], we allow inductive types to be polymorphic and dependent, as it is the case in CIC.

Definition 2 (Constructors) For $\mathcal{G} \subseteq \mathcal{F}$, let $\mathcal{R}_\mathcal{G}$ be the set of rules defining the symbols in $\mathcal{G}$, that is, the rules whose left-hand side is headed by a symbol in $\mathcal{G}$. The set of *free symbols* is $\mathcal{CF} = \{f \in \mathcal{F} \mid \mathcal{R}_{\{f\}} = \emptyset\}$. The set of *defined symbols* is $\mathcal{DF} = \mathcal{F} \backslash \mathcal{CF}$. The set of *constructors* of a free predicate symbol C is $\mathcal{C}o(C) = \{f \in \mathcal{F}^\star \mid \tau_f = (\vec{y} : \vec{U})C(\vec{v}) \text{ and } |\vec{y}| = \alpha_f\}$.

The constructors of C not only include the constructors in the usual sense but every defined symbol whose output type is C. For example, the symbols $0 : int$, $s : int \rightarrow int$, $p : int \rightarrow int$, $+ : int \rightarrow int \rightarrow int$ and $\times : int \rightarrow int \rightarrow int$ defined by the rules $s(p(x)) \rightarrow x$, $p(s(x)) \rightarrow x$ and others for $+$ and $\times$ are all constructors of the type *int* of integers.

Definition 3 (Inductive structure) An *inductive structure* is given by :

- a quasi-ordering $\geq_\mathcal{F}$ on $\mathcal{F}$, called *precedence*, whose strict part, $>_\mathcal{F}$, is well-founded,
- for each $C \in \mathcal{CF}^\Box$ such that $\tau_C = (\vec{x} : \vec{T})\star$, a set $Ind(C) \subseteq \{i \in \{1,..,\alpha_C\} \mid x_i \in \mathcal{X}^\Box\}$ of *inductive* positions,
- for each constructor c, a set $Acc(c) \subseteq \{1,..,\alpha_c\}$ of *accessible* positions.

The accessible positions allow the user to describe which patterns can be used for defining functions, and the inductive positions allow to describe the arguments on which the free predicate symbols should be monotone. This allows us to generalize the notion of positivity used in CIC.

Definition 4 (Positive and negative positions) The sets of *positive* positions $Pos^+(T)$ and *negative* positions $Pos^-(T)$ of a term T are mutually defined by induction on T as follows :

- $Pos^+(s) = Pos^+(F(\vec{t})) = Pos^+(X) = \{\varepsilon\}$,
- $Pos^-(s) = Pos^-(F(\vec{t})) = Pos^-(X) = \emptyset$,
- $Pos^\delta((x:V)W) = 1.Pos^{-\delta}(V) \cup 2.Pos^\delta(W)$,
- $Pos^\delta([x:V]W) = 1.Pos(V) \cup 2.Pos^\delta(W)$,
- $Pos^\delta(Vu) = 1.Pos^\delta(V) \cup 2.Pos(u)$,
- $Pos^\delta(VU) = 1.Pos^\delta(V)$,
- $Pos^+(C(\vec{t})) = \{\varepsilon\} \cup \bigcup\{i.Pos^+(t_i) \mid i \in Ind(C)\}$,
- $Pos^-(C(\vec{t})) = \bigcup\{i.Pos^-(t_i) \mid i \in Ind(C)\}$,

where $\delta \in \{\bot, +\}$, $\bot + = \bot$, $\bot\bot = +$.

For example, in $(x:A)B$, B occurs positively while A occurs negatively. Now, with the type *list* of polymorphic lists, A occurs positively in $list(A)$ iff $Ind(list) = \{1\}$.

Definition 5 (Admissible inductive structure) An inductive structure is *admissible* if, for all $C \in \mathcal{CF}^\Box$ with $\tau_C = (\vec{x} : \vec{T})\star$:

(I1) $\forall i \in Ind(C), v_i \in \mathcal{X}^\Box$,

and for all c with $\tau_c = (\vec{y} : \vec{U})C(\vec{v})$ and $j \in Acc(c)$:

(I2) $\forall i \in Ind(C), Pos(v_i, U_j) \subseteq Pos^+(U_j)$,
(I3) $\forall D \in \mathcal{CF}^\Box, D =_\mathcal{F} C \Rightarrow Pos(D, U_j) \subseteq Pos^+(U_j)$,
(I4) $\forall D \in \mathcal{CF}^\Box, D >_\mathcal{F} C \Rightarrow Pos(D, U_j) = \emptyset$,
(I5) $\forall F \in \mathcal{DF}^\Box, Pos(F, U_j) = \emptyset$,
(I6) $\forall X \in FV^\Box(U_j), \exists \iota_X \in \{1,..,\alpha_C\}, v_{\iota_X} = X$.

For example, with the type *list* of polymorphic lists, $Ind(list) = \{1\}$, $Acc(nil) = \{1\}$ and $Acc(cons) = \{1,2,3\}$ is an admissible inductive structure. If we add the type $tree : \star$ and the constructor $node : list(tree) \rightarrow tree$ with $Acc(node) = \{1\}$, we still have an admissible structure.

The condition (I6) means that the predicate-arguments of a constructor must be parameters of the

type they define. One can find a similar condition in the work of Walukiewicz [30] (called "$\star$-dependency") and in the work of Stefanova [27] (called "safeness").

On the other hand, there is no such explicit restriction in CIC. But the elimination scheme is typed in such a way that no very interesting function can be defined on a type not satisfying (I6). For example, consider the type of heterogeneous non-empty lists (we use the CIC syntax here) $listh = Ind(X : \star)\{C_1|C_2\}$ where $C_1 = (A : \star)(x : A)X$ and $C_2 = (A : \star)(x : A)$ $X \to X$. The typing rule for the non dependent elimination schema (Nodep$_{\star,\star}$) is :

$$\frac{\Gamma \vdash \ell : listh \quad \Gamma \vdash Q : \star \quad \forall i, \Gamma \vdash f_i : C_i\{listh, Q\}}{\Gamma \vdash Elim(\ell, Q)\{f_1|f_2\} : Q}$$

where $C_1\{listh, Q\} = (A : \star)(x : A)Q$ and $C_2\{listh, Q\} = (A : \star)(x : A)listh \to Q \to Q$. Since Q, f_1 and f_2 must be typable in Γ, the result of f_1 and f_2 cannot depend on A or on x. This means that it is possible to compute the length of such a list but not to use an element of the list.

Definition 6 (Primitive, basic and strictly positive predicates) A free predicate symbol C is :

- *primitive* if, for all $D =_{\mathcal{F}} C$, for all constructor d of type $\tau_d = (\vec{y} : \vec{U})D(\vec{w})$ and for all $j \in Acc(d)$, U_j is either of the form $E(\vec{t})$ with $E <_{\mathcal{F}} D$ and E basic, or of the form $E(\vec{t})$ with $E =_{\mathcal{F}} D$.
- *basic* if, for all $D =_{\mathcal{F}} C$, for all constructor d of type $\tau_d = (\vec{y} : \vec{U})D(\vec{w})$ and for all $j \in Acc(d)$, if $E =_{\mathcal{F}} D$ occurs in U_j then U_j is of the form $E(\vec{t})$.
- *strictly positive* if, for all $D =_{\mathcal{F}} C$, for all constructor d of type $\tau_d = (\vec{y} : \vec{U})D(\vec{w})$ and for all $j \in Acc(d)$, if $E =_{\mathcal{F}} D$ occurs in U_j then U_j is of the form $(\vec{z} : \vec{V})E(\vec{t})$ and no occurrence of $D' =_{\mathcal{F}} D$ occurs in $\vec{V}$.

For example, the type *list* of polymorphic lists is basic but not primitive. The type *listint* of lists of integers with the constructors $nilint : listint$ and $consint : int \to listint \to listint$ is primitive. And the type *ord* of Brouwer's ordinals with the constructors $0 : ord$, $s : ord \to ord$ and $lim : (nat \to ord) \to ord$ is strictly positive.

Although we do not explicitly forbid to have non-strictly positive predicate symbols, the admissibility conditions we are going to describe in the following subsections will not enable us to define functions on such a predicate. The same restriction applies on CIC while the system of Walukiewicz [30] is restricted to basic predicates and the λR-cube [1] or NDM [13] are restricted to primitive and non-dependent predicates. However, in the following, for lack of space, we will restrict our attention to basic predicates.

3.2 General Schema

The constructors of primitive predicates (remember that they include all symbols whose output type is a primitive predicate), defined by usual first-order rules, are easily shown to be strongly normalizing since the combination of first-order rewriting with $\to_\beta$ preserves strong normalization [8].

On the other hand, in the presence of higher-order rules, few techniques are known :

- Van de Pol [28] extended to the higher-order case the use of strictly monotone interpretations . This technique is very powerful but difficult to use in practice and has not been studied yet in type systems richer than the simply-typed λ-calculus.
- Jouannaud and Okada [21] defined a syntactic criterion, the General Schema, which extends primitive recursive definitions. This schema has been reformulated and enhanced to deal with definitions on strictly-positive types [6], to higher-order pattern-matching [3] and to richer type systems with object-level rewriting [1, 5].
- Jouannaud and Rubio [22] extended to the higher-order case the use of Dershowitz's recursive path ordering. The obtained ordering can be seen as a recursive version of the General Schema and has been extended by Walukiewicz [30] to the Calculus of Constructions with object-level rewriting.

Here, we present an extension of the General Schema defined in [5] to deal with type-level rewriting, the main novelty of our paper.

The General Schema is based on Tait and Girard's computability predicate technique [19] for proving the strong normalization of the simply-typed λ-calculus and system F. This technique consists in interpreting each type T by a set $[\![T]\!]$ of strongly normalizable terms, called *computable*, and in proving that $t \in [\![T]\!]$ whenever $\Gamma \vdash t : T$.

The idea of the General Schema is then to define, from a left-hand side of rule $f(\vec{l})$, a set of right-hand sides r that are computable whenever the l_i's are computable. This set is built from the variables of the left-hand side, called *accessible*, that are computable whenever the l_i's are computable, and is then closed by computability-preserving operations.

For the sake of simplicity, two sequences of arguments of a symbol f will be compared in a lexicographic manner. But it is possible to do these comparisons in a multiset manner or with a simple combination of lexicographic and multiset comparisons (see [4] for details).

Definition 7 (Accessibility) A pair $\langle u, U\rangle$ is *accessible* in a pair $\langle t, T\rangle$, written $\langle t, T\rangle \rhd_1 \langle u, U\rangle$, if $\langle t, T\rangle = \langle c(\vec{u}), C(\vec{v})\gamma\rangle$ and $\langle u, U\rangle = \langle u_j, U_j\gamma\rangle$ with c a constructor of type $\tau_c = (\vec{y} : \vec{U})C(\vec{v})$, $\gamma = \{\vec{y} \mapsto \vec{u}\}$ and $j \in Acc(c)$.

For example, in the definition of *app* previously given, A', x and ℓ are all accessible in $t = cons(A', x, \ell)$: $\langle t, list(A)\rangle \rhd_1 \langle A', \star\rangle$, $\langle t, list(A)\rangle \rhd_1 \langle x, A'\rangle$ and $\langle t, list(A)\rangle \rhd_1 \langle \ell, list(A')\rangle$.

Definition 8 (Derived type) Let t be a term of the form $l\sigma$ with $l = f(\vec{l})$ algebraic, $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$. Let $p \in Pos(l)$ with $p \neq \varepsilon$. The subterm $t|_p$ of t has a *derived type*, $\tau(t, p)$, defined as follows :
- if $p = i$ then $\tau(t, p) = T_i\gamma\sigma$,
- if $p = iq$ and $q \neq \varepsilon$ then $\tau(t, p) = \tau(t_i, q)$.

Definition 9 (Well-formed rule) Let $R = (l \to r, \Gamma, \rho)$ be a rule with $l = f(\vec{l})$, $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$. The rule R is *well-formed* if, for all $x \in dom(\Gamma)$, there is $i \leq \alpha_f$ and $p_x \in Pos(x, l_i)$ such that $\langle l_i, T_i\gamma\rangle \rhd_1^* \langle x, \tau(l, ip_x)\rangle$ and $\tau(l, ip_x)\rho = x\Gamma$.

Definition 10 (Computable closure) Let $R = (l \to r, \Gamma_0, \rho)$ be a rule with $l = f(\vec{l})$, $\tau_f = (\vec{x} : \vec{T})U$ et $\gamma = \{\vec{x} \mapsto \vec{l}\}$. The order $>$ on the arguments of f is the lexicographic extension of $\rhd_1^+$. The *computable closure* of R is the relation $\vdash_c$ defined by the rules of Figure 2.

Definition 11 (General Schema) A rule $(f(\vec{l}) \to r, \Gamma, \rho)$ with $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$ satisfies the *General Schema* if it is well-formed and $\Gamma \vdash_c r : U\gamma\rho$.

It is easy to check that the rules for *app* are well-formed and that $\Gamma \vdash_c cons(A, x, app(A, \ell, \ell')) : list(A)$. For example, we show that $\Gamma \vdash_c app(A, \ell, \ell') : list(A)$:

$$\dfrac{\dfrac{\Gamma \vdash_c \star : \Box}{\Gamma \vdash_c A : \star} \quad \dfrac{\dfrac{\dots}{\Gamma \vdash_c A : \star}}{\Gamma \vdash_c list(A) : \star} \quad \dfrac{\dots}{\Gamma \vdash_c \ell : list(A)} \quad \dfrac{\dots}{\Gamma \vdash_c \ell' : list(A)}}{\dfrac{\langle cons(A', x, \ell), list(A)\rangle > \langle \ell, list(A)\rangle}{\Gamma \vdash_c app(A, \ell, \ell')}}$$

3.3 Admissibility conditions

Definition 12 (Rewrite systems) Let $\mathcal{G}$ be a set of symbols. The *rewrite system* $(\mathcal{G}, \mathcal{R}_\mathcal{G})$ is :

- *algebraic* if :
  - $\mathcal{G}$ is made of predicate symbols or of constructors of primitive predicates,
  - all rules of $\mathcal{R}_\mathcal{G}$ have an algebraic right-hand side;
- *non-duplicating* if, for all $l \to r \in \mathcal{R}_\mathcal{G}$, no variable has more occurrences in r than in l;
- *primitive* if, for all rule $l \to r \in \mathcal{R}_\mathcal{G}$, r is of the form $[\vec{x} : \vec{T}]g(\vec{u})\vec{v}$ with g belonging to $\mathcal{G}$ or g being a primitive predicate symbol;
- *simple* if, for all $g(\vec{l}) \to r \in \mathcal{R}_\mathcal{G}$:
  - all the symbols occuring in $\vec{l}$ are free,
  - for all sequence of terms $\vec{t}$, at most one rule can apply at the top of $g(\vec{t})$,
  - for all rule $g(\vec{l}) \to r \in \mathcal{R}_\mathcal{G}$ and all $Y \in FV^\Box(r)$, there is a unique κ_Y such that $l_{\kappa_Y} = Y$;
- *positive* if, for all $l \to r \in \mathcal{R}_\mathcal{G}$ and all $g \in \mathcal{G}$, $Pos(g, r) \subseteq Pos^+(r)$;

Figure 2: Computable closure

$$\text{(acc)} \quad \dfrac{\Gamma_0 \vdash_c x\Gamma_0 : s \quad x \in dom^s(\Gamma_0)}{\Gamma_0 \vdash_c x : x\Gamma_0}$$

$$\text{(ax)} \quad \dfrac{}{\Gamma_0 \vdash_c \star : \Box}$$

$$\text{(symb}^<\text{)} \quad \dfrac{\begin{array}{c} g \in \mathcal{F}_n^s,\ \tau_g = (\vec{y} : \vec{U})V,\ \gamma = \{\vec{y} \mapsto \vec{u}\} \\ g <_\mathcal{F} f \quad \Gamma \vdash_c \tau_g : s \quad \forall i,\ \Gamma \vdash_c u_i : U_i\gamma \end{array}}{\Gamma \vdash_c g(\vec{u}) : V\gamma}$$

$$\text{(symb}^=\text{)} \quad \dfrac{\begin{array}{c} g \in \mathcal{F}_n^s,\ \tau_g = (\vec{y} : \vec{U})V,\ \gamma = \{\vec{y} \mapsto \vec{u}\} \\ g =_\mathcal{F} f \quad \Gamma \vdash_c \tau_g : s \quad \forall i,\ \Gamma \vdash_c u_i : U_i\gamma \\ \langle \vec{l}, \vec{T}\gamma_0\rangle > \langle \vec{u}, \vec{U}\gamma\rangle \end{array}}{\Gamma \vdash_c g(\vec{u}) : V\gamma}$$

$$\text{(var)} \quad \dfrac{\Gamma \vdash_c T : s \quad x \in \mathcal{X}^s \setminus FV(l)}{\Gamma, x : T \vdash_c x : T}$$

$$\text{(weak)} \quad \dfrac{\Gamma \vdash_c t : T \quad \Gamma \vdash_c U : s \quad x \in \mathcal{X}^s \setminus FV(l)}{\Gamma, x : U \vdash_c t : T}$$

$$\text{(prod)} \quad \dfrac{\Gamma \vdash_c T : s \quad \Gamma, x : T \vdash_c U : s'}{\Gamma \vdash_c (x : T)U : s'}$$

$$\text{(abs)} \quad \dfrac{\Gamma, x : T \vdash_c u : U \quad \Gamma \vdash_c (x : T)U : s}{\Gamma \vdash_c [x : T]u : (x : T)U}$$

$$\text{(app)} \quad \dfrac{\Gamma \vdash_c t : (x : U)V \quad \Gamma \vdash_c u : U}{\Gamma \vdash_c tu : V\{x \mapsto u\}}$$

$$\text{(conv)} \quad \dfrac{\Gamma \vdash_c t : T \quad T \downarrow^* T' \quad \Gamma \vdash_c T' : s'}{\Gamma \vdash_c t : T'}$$

- *recursive* if all the rules of $\mathcal{R}_\mathcal{G}$ satisfy the General Schema;
- *safe* if, for all $(g(\vec{l}) \to r, \Gamma, \rho) \in \mathcal{R}_\mathcal{G}$ with $\tau_g = (\vec{x} : \vec{T})\ U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$:
  - for all $X \in FV^\square(\vec{T}U)$, $X\gamma\rho \in dom^\square(\Gamma)$,
  - for all $X, X' \in FV^\square(\vec{T}U)$, $X\gamma\rho = X'\gamma\rho \Rightarrow X = X'$.

Definition 13 (Admissible CAC) A CAC is *admissible* if :

(A1) $\to = \to_\mathcal{R} \cup \to_\beta$ is confluent;
(A2) its inductive structure is admissible;
(A3) $(\mathcal{DF}^\square, \mathcal{R}_{\mathcal{DF}^\square})$ is either :
  - primitive,
  - simple and positive,
  - simple and recursive;

(A4) there is a partition $\mathcal{F}_a \uplus \mathcal{F}_{na}$ of $\mathcal{DF}$ (*algebraic* and *non-algebraic* symbols) such that :
  - $(\mathcal{F}_a, \mathcal{R}_{\mathcal{F}_a})$ is algebraic, non-duplicating and strongly normalizing,
  - no symbol of $\mathcal{F}_{na}$ occurs in the rules of $\mathcal{R}_{\mathcal{F}_a}$,
  - $(\mathcal{F}_{na}, \mathcal{R}_{\mathcal{F}_{na}})$ is safe and recursive.

The simplicity condition in (A3) extends to the case of rewriting the restriction in CIC of strong elimination to "small" inductive types, that is, to the types whose constructors have no predicate-arguments except the parameters of the type.

The safeness condition in (A4) means that one cannot do pattern-matching or equality tests on predicate-arguments that are necessary for typing other arguments. In her extension of HORPO to the Calculus of Constructions, Walukiewicz requires similar conditions [30].

The non-duplication condition in (A4) ensures the modularity of the strong normalization. Indeed, in general, the combination of two strongly normalizing rewrite systems is not strongly normalizing.

Now, for proving (A1), one can use the following result of van Oostrom [29] (remember that $\mathcal{R} \cup \beta$ can be seen as a CRS [23]) : the combination of two confluent left-linear CRS's having no critical pairs between each other is confluent. So, since $\to_\beta$ is confluent and $\mathcal{R}$ and β cannot have critical pairs between each other, if $\mathcal{R}$ is left-linear and confluent then $\to_\mathcal{R} \cup \to_\beta$ is confluent. Therefore, our conditions (S1) to (S5) are very useful to eliminate the non-linearities due to typing reasons.

We can now state our main result. You can find a detailed proof in [4].

Theorem 14 (Strong normalization) Any admissible CAC is strongly normalizing.

The proof is based on Coquand and Gallier's extension to the Calculus of Constructions [9] of Tait and Girard's computability predicate technique [19]. As explained before, the idea is to define an interpretation for each type and to prove that each well-typed term belongs to the interpretation of its type.

The main difficulty is to define an interpretation for predicate symbols that is invariant by reduction, a condition required by the type conversion rule (conv).

Thanks to the positivity conditions, the interpretation of a free predicate symbol can be defined as the least fixpoint of a monotone function over the lattice of computability predicates.

For the defined predicate symbols, it depends on the kind of system $(\mathcal{DF}^\square, \mathcal{R}_{\mathcal{DF}^\square})$ is. If it is primitive then we simply interpret it as the set of strongly normalizable terms. If it is positive then, thanks to the positivity condition, we can interpret it as a least fixpoint. Finally, if it is recursive then we can define its interpretation recursively, the General Schema providing a well-founded definition.

4 Examples

4.1 Calculus of Inductive Constructions

We are going to see that we can apply our strong normalization theorem to a sub-system of CIC [26] by translating it into an admissible CAC. The first complete proof of strong normalization of CIC (with strong elimination) is due to Werner [31] who, in addition, considers η-reductions in the type conversion rule.

In CIC, one has strictly-positive inductive types and the corresponding induction principles. We recall the syntax and the typing rules of CIC but, for the sake of simplicity, we will restrict our attention to basic inductive types and non-dependent elimination schemas. For a complete presentation, see [4].

- Inductive types are denoted by $Ind(X : A)\{\vec{C}\}$ where the C_i's are the types of the constructors. The term A must be of the form $(\vec{x} : \vec{A})\star$ and the C_i's of the form $(\vec{z} : \vec{B})X\vec{m}$.
- The i-th constructor of an inductive type I is denoted by $Constr(i, I)$.
- Recursors are denoted by $Elim(I, Q, \vec{a}, c)$ where I is the inductive type, Q the type of the result, $\vec{a}$ the arguments of I and c a term of type $I\vec{a}$.

The typing rules for these constructions are given in Figure 3. The rules for the other constructions are the same as for the Calculus of Constructions.

If $C_i = (\vec{z} : \vec{B})X\vec{m}$ then $C_i\{I, Q\}$ denotes $(\vec{z} : \vec{B})(\vec{z'} : \vec{B}\{X \mapsto Q\})\ Q\vec{m}$. The reduction relation associated to

Figure 3: Typing rules of CIC

$$(\mathrm{Ind}_\star)\quad \frac{\forall i,\ \Gamma, X : A \vdash C_i : \star}{\Gamma \vdash Ind(X:A)\{\vec{C}\} : A}$$

$$(\mathrm{Constr})\quad \frac{\Gamma \vdash I = Ind(X:A)\{\vec{C}\} : A}{\Gamma \vdash Constr(i,I) : C_i\{X \mapsto I\}}$$

$$(\mathrm{Nodep}_{\star,s})\quad \frac{\Gamma \vdash c : I\vec{a} \quad \Gamma \vdash Q : (\vec{x} : \vec{A})s \quad \forall i,\ \Gamma \vdash f_i : C_i\{I, Q\}}{\Gamma \vdash Elim(I, Q, \vec{a}, c)\{\vec{f}\} : Q\vec{a}}$$

Elim is called *ι-reduction* and is defined as follows :

$$Elim(I, Q, \vec{a}, Constr(i, I')\,\vec{b})\{\vec{f}\} \to_\iota f_i\,\vec{b}\,\vec{b'}$$

where, if $C_i = (\vec{z} : \vec{B})X\vec{m}$, then $b'_j = Elim(I, Q, \vec{a'}, b_j)$ if $B_j = X\vec{a'}$, and $b'_j = b_j$ otherwise.

Now, we consider the sub-system CIC$^-$ obtained by applying the following restrictions :

- In the typing rules ($\mathrm{Ind}_\star$) and (Constr), we assume that Γ is empty since, in CAC, the types of the symbols must be typable in the empty environment.
- In the rule ($\mathrm{Nodep}_{\star,\star}$) (the one for weak elimination), we require Q to be typable in the empty environment.
- In the rule ($\mathrm{Nodep}_{\star,\Box}$) (the one for strong elimination), instead of requiring $\Gamma \vdash Q : (\vec{x} : \vec{A})\Box$ which is not possible in the Calculus of Constructions since $\Box$ is not typable, we require Q to be a closed term of the form $[\vec{x} : \vec{A}]K$ with K of the form $(\vec{y} : \vec{U})\star$.
- We assume that every inductive type satisfies (I6).

Theorem 15 CIC$^-$ can be translated into an admissible CAC, hence is strongly normalizing.

We define the translation $\langle\ \rangle$ by induction on the size of terms :

- Let $I = Ind(X:A)\{\vec{C}\}$. We define $\langle I\rangle = [\vec{x} : \langle\vec{A}\rangle]\ Ind_I(\vec{x})$ where Ind_I is a symbol of type $(\vec{x} : \langle\vec{A}\rangle)\star$.
- By assumption, $C_i = (\vec{z} : \vec{B})X\vec{m}$. We define $\langle Constr(i,I)\rangle = [\vec{z} : \vec{B}]Constr^i_I(\vec{z})$ where $Constr^i_I$ is a symbol of type $(\vec{z} : \langle\vec{B}\rangle)Ind_I(\langle\vec{m}\rangle)$.
- Let $T_i = C_i\{I, Q\}$. If $Q = [\vec{x} : \vec{A}]K$ then we define $\langle Elim(I, Q, \vec{a}, c)\{\vec{f}\}\rangle = SElim^Q_I(\langle\vec{f}\rangle, \langle\vec{a}\rangle, \langle c\rangle)$ where $SElim^Q_I$ is a symbol of type $(\vec{f} : \langle\vec{T}\rangle)\ (\vec{x} : \langle\vec{A}\rangle)\ \langle K\rangle$. Otherwise, we define $\langle Elim(I, Q, \vec{a}, c)\{\vec{f}\}\rangle = WElim_I(\langle Q\rangle, \langle\vec{f}\rangle, \langle\vec{a}\rangle, \langle c\rangle)$ where $WElim_I$ is a symbol of type $(Q : \langle A\rangle)(\vec{f} : \langle\vec{T}\rangle)(\vec{x} : \langle\vec{A}\rangle)\ \langle Q\rangle\vec{x}$.
- The other terms are defined recursively ($\langle uv\rangle = \langle u\rangle\langle v\rangle, \ldots$).

The ι-reduction is translated by the following rules :

$$SElim^Q_I(\vec{f}, \vec{a}, Constr^i_I(\vec{b})) \ \to\ f_i\,\vec{b}\,\vec{b'}$$
$$WElim_I(Q, \vec{f}, \vec{a}, Constr^i_I(\vec{b})) \ \to\ f_i\,\vec{b}\,\vec{b'}$$

where, if $C_i = (\vec{z} : \vec{B})X\vec{m}$, then $b'_j = SElim^Q_I(\vec{f}, \vec{a'}, b_j)$ (or $WElim_I(Q, \vec{f}, \vec{a'}, b_j)$) if $B_j = X\vec{a'}$, and $b'_j = b_j$ otherwise.

Now, we are left to check the admissibility :

(A1) $\to_{\beta\iota}$ is orthogonal, hence confluent [29].

(A2) The inductive structure defined by $I <_{\mathcal{F}} J$ if I is a subterm of J, $Ind(Ind_I) = \emptyset$, $Acc(Constr^i_I) = \{1, .., |\vec{z}|\}$ if $C_i = (\vec{z} : \vec{B})X\vec{m}$, is admissible.

(A3) The rules defining the strong recursors form a simple (they are defined by case on each constructor and only for small inductive types) and recursive rewrite system (they satisfy the General Schema).

(A4) The rules defining the recursors form a safe (except for the constructor, all the arguments are distinct variables) and recursive rewrite system (they satisfy the General Schema).

4.2 Natural Deduction Modulo

NDM for first-order logic [12] can be presented as an extension of Natural Deduction with the additional inference rule :

$$\frac{\Gamma \vdash P}{\Gamma \vdash Q} \quad \text{if } P \equiv Q$$

where $\equiv$ is a congruence relation on propositions. This is a powerful extension of first-order logic since both higher-order logic and set theory with a comprehension symbol can be described in this framework (by using explicit substitutions).

In [13], Dowek and Werner study the termination of cut-elimination in the case where $\equiv$ is induced by a confluent and weakly-normalizing rewrite system. In particular, they prove the termination in two general cases : when the rewrite system is positive and when it is quantifier-free. In [14], they provide an example of confluent and weakly normalizing rewrite system for which cut-elimination is not terminating. The problem comes from the fact that the elimination rule for $\forall$ introduces a substitution :

$$\frac{\Gamma \vdash \forall x.P(x)}{\Gamma \vdash P(t)}$$

Thus, when a predicate symbol is defined by a rule whose right-hand side contains quantifiers, its combi-

nation with β may not preserve normalization. Therefore, a criterion for higher-order rewriting is needed.

Since NDM is a CAC (we can define the logical connectors as inductive types), we can compare in more details the conditions of [13] with our conditions.

(A1) In [13], only $\rightarrow_{\mathcal{R}}$ is required to be confluent. In general, this is not sufficient for having the confluence of $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$. However, if $\mathcal{R}$ is left-linear then $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$ is confluent [29].

(A2) NDM types are primitive and form an admissible inductive structure if we take them equivalent in the relation $\leq_{\mathcal{F}}$.

(A3) In [13], the termination of cut-elimination is proved in two general cases : when $(\mathcal{DF}^{\square}, \mathcal{R}_{\mathcal{DF}^{\square}})$ is quantifier-free and when it is positive. Quantifier-free rewrite systems are primitive. So, in this case, (A3) is satisfied. In the positive case, we require that left-hand sides are made of free symbols and that at most one rule can apply at the top of a term. On the other hand, we provide a new case : $(\mathcal{DF}^{\square}, \mathcal{R}_{\mathcal{DF}^{\square}})$ can be simple and recursive.

(A4) Quantifier-free rules are algebraic and rules with quantifiers are not. In [13], these two kinds of rules are treated in the same way but the counter-example given in [14] shows that they should not. In CAC, we require that the rules with quantifiers satisfy the General Schema.

Theorem 16 A NDM system satisfying (A1), (A3) and (A4) is admissible, hence strongly normalizing.

4.3 CIC + Rewriting

As a combination of the two previous applications, our work shows that the extension of CIC$^-$ with user-defined rewrite rules, even at the predicate-level, is sound if these rules follow our admissibility conditions.

As an example, we consider simplification rules on propositions that are not definable in CIC. Assume that we have the symbols $\vee : \star \rightarrow \star \rightarrow \star$, $\wedge : \star \rightarrow \star \rightarrow \star$, $\neg : \star \rightarrow \star$, $\bot : \star$, $\top : \star$, and the rules :

$$\begin{array}{lll} \top \vee P \rightarrow \top & \bot \wedge P \rightarrow \bot & \neg\top \rightarrow \bot \\ P \vee \top \rightarrow \top & P \wedge \bot \rightarrow \bot & \neg\bot \rightarrow \top \end{array}$$

$$\neg(P \wedge Q) \rightarrow \neg P \vee \neg Q \qquad \neg(P \vee Q) \rightarrow \neg P \wedge \neg Q$$

The predicate constructors $\vee$, $\wedge$, ... are all primitive. The rewrite system is primitive, algebraic, strongly normalizing and confluent (this can be automatically proved by CiME [16]). Since it is left-linear, its combination with $\rightarrow_{\beta}$ is confluent [29]. Therefore, it is an admissible CAC. But it lacks many other rules [20] which requires rewriting modulo associativity and commutativity, an extension we leave for future work.

5 Conclusion

We have defined an extension of the Calculus of Constructions by functions and predicates defined with rewrite rules. The main contributions of our work are the following :

- We consider a general notion of rewriting at the predicate-level which generalizes the "strong elimination" of the Calculus of Inductive Constructions [26, 31]. For example, we can define simplification rules on propositions that are not definable in CIC.
- We consider general syntactic conditions, including confluence, that ensure the strong normalization of the calculus. In particular, these conditions are fulfilled by two important systems : a sub-system of the Calculus of Inductive Constructions which is the basis of the proof assistant Coq [17], and the Natural Deduction Modulo [12, 13] a large class of equational theories.
- We use a more general notion of constructor which allows pattern-matching on defined symbols and equations among constructors.
- We relax the usual conditions on rewrite rules for ensuring the subject reduction property. By this way, we can eliminate some non-linearities in left-hand sides of rules and ease the confluence proof.

6 Directions for future work

- In our conditions, we assume that the predicate symbols defined by rewrite rules containing quantifiers ("non-primitive" predicate symbols) are defined by pattern-matching on free symbols only ("simple" systems). It would be nice to be able to relax this condition.
- Another important assumption is that the reduction relation $\rightarrow_{=} = \rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$ must be confluent. We will try to find sufficient conditions on $\mathcal{R}$ in order to get the confluence of $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$. In the simply-typed λ-calculus, if $\mathcal{R}$ is a first-order rewrite system then the confluence of $\mathcal{R}$ is a sufficient condition [7]. But few results are known in the case of a richer type system or of higher-order rewriting.
- Finally, we expect to extend this work with rewriting modulo some useful equational theories like associativity and commutativity, and also by allowing η-reductions in the type conversion rule.

Acknowledgments : I would like to thank Daria Walukiewicz, Gilles Dowek, Jean-Pierre Jouannaud and Christine Paulin for useful comments on previous versions of this work.

References

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

[2] H. Barendregt. Lambda calculi with types. In S. Abramski, D. Gabbay, and T. Maibaum, editors, *Handbook of logic in computer science*, volume 2. Oxford University Press, 1992.

[3] F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. of RTA'00*, LNCS 1833.

[4] F. Blanqui. *Théorie des Types et Réécriture (Type Theory and Rewriting)*. PhD thesis, Université Paris-Sud (France), 2001. Available at `http://www.lri.fr/~blanqui`. An english version will be available soon.

[5] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In *Proc. of RTA'99*, LNCS 1631.

[6] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 277, 2001.

[7] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of LICS'88*, IEEE Computer Society.

[8] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3–28, 1991.

[9] T. Coquand and J. Gallier. A proof of strong normalization for the Theory of Constructions using a Kripke-like interpretation, 1990. Paper presented at the 1st Int. Work. on Logical Frameworks but not published in the proceedings. Available at `ftp://ftp.cis.upenn.edu/pub/papers/gallier/sntoc.dvi.Z`.

[10] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[11] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. North-Holland, 1990.

[12] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Technical Report 3400, INRIA Rocquencourt (France), 1998.

[13] G. Dowek and B. Werner. Proof normalization modulo. In *Proc. of TYPES'98*, LNCS 1657.

[14] G. Dowek and B. Werner. An inconsistent theory modulo defined by a confluent and terminating rewrite system, 2000. Available at `http://pauillac.inria.fr/~dowek/`.

[15] C. Kirchner *et al.* ELAN, 2000. Available at `http://elan.loria.fr/`.

[16] C. Marché *et al.* CiME, 2000. Available at `http://www.lri.fr/~demons/cime.html`.

[17] C. Paulin *et al.* *The Coq Proof Assistant Reference Manual Version 6.3.1.* INRIA Rocquencourt (France), 2000. Available at `http://coq.inria.fr/`.

[18] H. Geuvers, R. Nederpelt, and R. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[19] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[20] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25:255–300, 1985.

[21] J.-P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[22] J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proc. of LICS'99*, IEEE Computer Society.

[23] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems : introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[24] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Napoli, Italy, 1984.

[25] N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, United States, 1987.

[26] C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *Proc. of TLCA'93*, LNCS 664.

[27] M. Stefanova. *Properties of Typing Systems*. PhD thesis, Nijmegen University (Netherlands), 1998.

[28] J. van de Pol. *Termination of higher-order rewrite systems*. PhD thesis, University of Utrecht, Nederlands, 1994.

[29] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Netherlands, 1994.

[30] D. Walukiewicz. Termination of rewriting in the Calculus of Constructions. In *Proc. of LFM'00*.

[31] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, France, 1994.