Inductive Definitions in the system Coq Rules and Properties

Christine Paulin-Mohring [*]

LIP-IMAG, URA CNRS 1398
Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon cedex 07, France
e-mail : cpaulin@lip.ens-lyon.fr

Abstract. In the pure Calculus of Constructions, it is possible to represent data structures and predicates using higher-order quantification. However, this representation is not satisfactory, from the point of view of both the efficiency of the underlying programs and the power of the logical system. For these reasons, the calculus was extended with a primitive notion of inductive definitions [8]. This paper describes the rules for inductive definitions in the system Coq. They are general enough to be seen as one formulation of adding inductive definitions to a typed lambda-calculus. We prove strong normalization for a subsystem of Coq corresponding to the pure Calculus of Constructions plus Inductive Definitions with only weak eliminations.

1 Introduction

1.1 Motivations

Several proof environments suitable for mechanizing mathematics and program development [10, 5] are based on the "Curry-Howard correspondence" between natural deduction proofs and typed functional programs. Such proof tools are used interactively and consequently aims at providing rules as natural as possible and avoid tedious encoding. Such a motivation was the starting point for an extension of the pure Calculus of Constructions with primitive inductive definitions.

The (Pure) Calculus of Constructions extends the powerful polymorphic programming language F_ω with dependent types and allows reasoning about programs. Its main advantage is to be a "closed system" where mathematical and computational notions can be internally represented using higher-order quantification. But this representation is not satisfactory from both the computational and the logical points of view. This leads to a proposition for an extension of the Calculus of Constructions with inductive definitions as first-class objects [8]. The rules for these definitions follow the point of view of Martin-Löf's type theory. From a specification of the introduction rules for a new inductive definition, we generate a dependent elimination with new computational rules. This elimination for natural numbers corresponds both to the construction of functions following a primitive recursive scheme

[*] This research was partly supported by ESPRIT Basic Research Action "Types for Proofs and Programs" and by Programme de Recherche Coordonnées "Programmation avancée et Outils pour l'Intelligence Artificielle".

and to proofs by induction. Our extension of the Calculus of Constructions with Inductive Definitions, unlike Martin-Löf's type theory, still preserve the property for the system to be closed.

There exists an implementation of this extension, namely the system Coq [10] developed in the Formel project at Inria-Rocquencourt and ENS Lyon. The mechanism for inductive definitions has proved to be really useful for the development of examples but its meta-theory is not yet established. The purpose of this paper is to give a precise description of the rules used in the system and state what we know about its properties. In particular we shall prove the strong normalization for a subsystem of Coq, corresponding to the pure Calculus of Constructions plus Inductive Definitions with only weak eliminations. For this, we use a similar result obtained by Ph. Audebaud [1, 2] for an extension of the Calculus of Constructions with a fixpoint operator.

The paper is organized as follow. The remaining part of this section describes the drawbacks of the impredicative coding of inductive definitions in the pure Calculus of Constructions and gives an intuitive idea, using the example of natural numbers, of the rules for elimination we shall introduce. Part 2 introduces schematic rules for adding inductive definitions to a typed lambda-calculus. Part 3 gives the rules used in the system Coq and examples of inductive definitions. In part 4 the strong normalization of a subsystem F_ω^{Ind} of Coq is established. In part 5 we shall discuss our choices and make comparisons with other systems.

1.2 Impredicative inductive definitions

In [4, 21, 22] a systematic way to generate a representation of an inductive definition from a description of the generative rules for it was described. But this representation is not really adequate. We list some problems :

- We can represent a type of boolean with two elements true and false but the fact that true and false are not equal is not provable in the system.
- We can represent a type of natural numbers but the induction principle is not provable.
- We can define all primitive recursive functions on natural numbers. That is given two terms x and g, we can find h such that $(h\ 0) = x$ and $(h\ n{+}1) = (g\ n\ (h\ n))$ holds internally for a closed natural number n (and consequently are provable). But if n is not closed then the proposition $(h\ n{+}1) = (g\ n\ (h\ n))$ (using Leibniz's equality) is provable using an induction over n but is not an internal reduction rule. Moreover for n closed, the reduction of $(h\ n+1)$ into $(g\ n\ (h\ n))$ can take a time proportional to n. This corresponds to the well-known problem of the predecessor function (the predecessor of $n+1$ is computed in n steps) and is a problem inherent to the representation of natural numbers with what is known as Church's numerals.
- Some structures represented with an impredicative coding (like the products of two types) can contain more elements (closed normal terms) than the one built from the constructors of the type. It is explained for instance in [21].

Some points concerning the representation of inductive definitions in an impredicative type theory are reflected in our rules for primitive inductive definitions. For instance an inductive definition makes sense in any context (and not just at toplevel); the names of the type and of its constructors are not relevant for the type conversion rules (so two types with the same specification will be equal). We use a uniform rule for the definitions of disjunction, conjunction, types and relations.

1.3 The case of natural numbers

The problem with the inductive definitions in impredicative systems comes from the weakness of the elimination scheme. The elimination scheme is intended to make use of the assumption that the inductive type is the smallest set generated by the introduction rules.

The specification of the natural numbers is a set nat with two constructors for zero and the successor function : 0 of type nat and S of type $nat \rightarrow nat$. Each "mathematical" natural number n can be represented as a term $(S^n\ 0)$. The elimination scheme internalizes the fact that the type of natural numbers only contains elements representing $(S^n\ 0)$ for some n.

The first problem is to be able to represent enough functions on natural numbers. For this, we need a special kind of recursive definitions. Theoretically, the representation of functions H on nat iteratively defined by equations $H(0) = x$, $H(S\ n) = f(H(n))$ combined with a notion of product is enough to get many interesting functions (in F_ω for instance, all recursive functions provably total in higher-order arithmetic).

But this representation is not always efficient. For instance, there is no representation in F_ω of the predecessor function which computes the predecessor of $(S\ n)$ in a constant number of reductions.

The predecessor function can be efficiently obtained in a ML-like language where the basic operation for the elimination of a concrete type is the definition by pattern. In this framework, we can represent a function H which satisfies the equation $H(0) = x$, $H(S\ n) = f(n)$. In ML, a general fixpoint is available to encode recursive calls.

To keep the property of strong normalization, we do not allow a general fixpoint and restrict the use of recursion. We want a direct representation of functions H which satisfies the properties $H(0) = x$, $H(S\ n) = f(n, H(n))$. This can be done by the introduction of a recursor operator $R(C, x, f, n)$ such that if C is a type, x of type C, f of type $nat \rightarrow C \rightarrow C$ and n of type nat then $R(C, x, f, n)$ has type C. Furthermore $R(C, x, f, 0)$ behaves like x and $R(C, x, f, (S\ n))$ behaves like $(f\ n\ (R(C, x, f, n))$. If $H(p)$ denotes $R(C, x, f, n)$ the equality $H(S\ n) = (f\ n\ H(n))$ will hold not only for closed terms n but also for a variable.

For each inductive definition we shall introduce the analogous of the R operator for natural numbers. The term $R(C, x, f, n)$ will be written as $\mathsf{Elim}(n, C)\{x|f\}$ in this paper, it corresponds to the notation `(<C>Match n with x f)` in Coq.

Dependent elimination. A system like Coq is intended to program functions but also to reason about programs. It is natural to want an induction principle :

$$\forall n : nat. \forall P.(P\ 0) \rightarrow (\forall u : nat.(P\ u) \rightarrow (P\ (S\ u))) \rightarrow (P\ n)$$

This proposition can be used to prove logical properties of programs. Now, in the paradigm of proofs as programs, we interpret (intuitionistic) proofs of a property P

as correct programs w.r.t. the specification represented by P. If n is a natural number, if x is a correct program for the specification $(P\ 0)$ and f a correct program for the specification $\forall u : nat.(P\ u) \rightarrow (P\ (S\ u))$ then we can built using the induction principle a correct program w.r.t. $(P\ n)$. It is easy to see that this program should behave like the program $R(C, x, f, n)$ built from the recursor R. Obviously the recursor can be obtained from the induction principle just taking a constant predicate $(P\ n) \equiv C$. This suggests that we need only one operator, with the induction principle as type and which obeys the same reduction rules as the recursor operator. This is the principle behind the rules in Martin-Löf's Intuitionistic Type Theory [18]. We shall also follow the same ideas for the system Coq.

Strong eliminations When we have a concrete inductive structure like the natural numbers, we may want to define a property (or a type) by induction over the structure of the natural number. This possibility is called "strong elimination" because we are building a property (or equivalently a type) by computation over a program. For instance, we could define a property P on natural numbers such that $(P\ 0)$ is a true proposition (for instance $(A : \mathrm{Set})A \rightarrow A$) and $(P\ (S\ n))$ is the absurdity proposition $(A : \mathrm{Set})A$. Then assuming $0 = (S\ n)$, because $(P\ 0)$ is true, we have a proof of $(P\ (S\ n))$ which is the absurdity. This extension corresponds to a strong modification of the underlying typed lambda-calculus.

2 Rules for inductive definitions

In this section we introduce schematic rules (parameterized by sorts) for inductive definitions and illustrate them on the example of the inductive definition of lists of a given length.

2.1 Notations

We are interested in an extension of the pure Calculus of Constructions, but actually our proposition can be defined somehow independently of the underlying Generalized Type System (GTS). The systems we are studying in that paper extends a pure type system with new constructors for inductive definitions.

Generalized type systems We use the now standard presentation of typed λ-calculi as functional GTS [3, 12]. The set of (pseudo) terms contains the following constructions :

$$t ::= s \mid x \mid (x:t)t \mid [x:t]t \mid (t\ t)$$

with s ranging over the set $\mathcal{S}$ of sorts. For the pure Calculus of Constructions, we shall use $\mathcal{S} = \{\mathrm{Set}, \mathrm{Type}_{\mathcal{S}}\}$ corresponding respectively to the star and the square in Barendregt's λ-cube. The set of axioms is $\mathcal{A} = \{\mathrm{Set} : \mathrm{Type}_{\mathcal{S}}\}$. All rules for product are allowed : the set of rules is $\mathcal{R} = \mathcal{S} \times \mathcal{S}$.

We write $M[x \leftarrow N]$ to denote the substitution of the term N to free occurrences of the variable x in M. The notation $M \rightarrow N$ is an abbreviation for $(x : M)N$ whenever x does not occur in N. The arrow symbol associates to the right and the application associates to the left.

We say that a term is a *type* if it is correctly typed and that its type is a sort.

Vectorial notations We introduce vectorial notations to write parametric rules.

Construction. Let $\boldsymbol{x}$ and $\boldsymbol{A}$ be two possibly empty sequences of resp. variables and terms with the same length. Let M be a term, we define the terms $(\boldsymbol{x} : \boldsymbol{A})M$, $[\boldsymbol{x} : \boldsymbol{A}]M$ and $(M\ \boldsymbol{A})$ for an arbitrary M by induction over the structure of the sequences $\boldsymbol{x}$ and $\boldsymbol{A}$.

- If $\boldsymbol{x}$ and consequently $\boldsymbol{A}$ are empty then $(\boldsymbol{x} : \boldsymbol{A})M = [\boldsymbol{x} : \boldsymbol{A}]M = (M\ \boldsymbol{A}) = M$.
- if $\boldsymbol{x} = x, \boldsymbol{x'}$ and $\boldsymbol{A} = A, \boldsymbol{A'}$ then $(\boldsymbol{x} : \boldsymbol{A})M = (x{:}A)(\boldsymbol{x'} : \boldsymbol{A'})M$, $[\boldsymbol{x} : \boldsymbol{A}]M = [x{:} A][\boldsymbol{x'} : \boldsymbol{A'}]M$ and $(M\ \boldsymbol{A}) = ((M\ A)\ \boldsymbol{A'})$.

Decomposition. We use the same notations to describe the decomposition of a term. Let P be a term, we shall write $P \equiv [\boldsymbol{x} : \boldsymbol{A}]M$ (resp. $P \equiv (\boldsymbol{x} : \boldsymbol{A})M$, resp. $P \equiv (M\ \boldsymbol{A})$) to define $\boldsymbol{x}$ and $\boldsymbol{A}$ as maximal sequences of terms such that the equality $P = [\boldsymbol{x} : \boldsymbol{A}]M$ (resp. $P = (\boldsymbol{x} : \boldsymbol{A})M$, resp. $P = (M\ \boldsymbol{A})$) holds.

For instance, a type in normal form can uniquely be written as $(\boldsymbol{x} : \boldsymbol{A})s$ or $(\boldsymbol{x}{:}\boldsymbol{A})(X\ \boldsymbol{a})$ with X a variable and s a sort.

2.2 Preliminary Definitions

Definition 1 Arity. An *arity of sort s* is a term generated by the following syntax : $Ar ::= s \mid (x{:}M)Ar$. We denote by *arity(A, s)* the property A is an arity of sort s.

Definition 2 Strictly positive types. Let A be an arity and X be of type A, the terms P which are *strictly positive* w.r.t. to X are generated by the syntax : $Pos ::= X \mid (Pos\ m) \mid (x{:}M)Pos$ with the restriction that X does not occur in M and m. We write *strict_positive(P, X)* the property P is strictly positive w.r.t. X.

A strictly positive well-formed type can be written as as $(\boldsymbol{x} : \boldsymbol{M})(X\ \boldsymbol{m})$ with the restriction that X does not occur in any term of $\boldsymbol{M}$ or $\boldsymbol{m}$.

Definition 3 Forms of constructor. Let A be an arity and X be of type A, the terms C which are a *form of constructor* w.r.t. X are generated by the syntax : $Co ::= X \mid (Co\ m) \mid P \rightarrow Co \mid (x{:}M)Co$ with the restriction that *strict_positive(P, X)* and X does not occur in M or m. We say that C is a *type of constructor of* X if furthermore, C is a type.
We write *constructor(C, X)* the property C is a form of constructor w.r.t. X.

A well-formed form of constructor can be written as $C \equiv (\boldsymbol{z} : \boldsymbol{C})(X\ \boldsymbol{a})$. A strictly positive type is a particular case of type of constructors, which we call a *non-recursive* type of constructor. A type of constructor which is not a strictly positive type is called *recursive.*

2.3 Inductive Operators

We introduce a new pseudo-term for inductive definitions $t ::= \mathsf{Ind}(x{:}t)\{\boldsymbol{t}\}$ with $\boldsymbol{t}$ a possibly empty list of terms, written as $t_1 | \ldots | t_n$.

In the expression $\mathsf{Ind}(x:t)\{\boldsymbol{t}\}$, x is a bound variable. The typing rule for this operator is, with n an arbitrary number, possibly equal to 0 :

$$\frac{arity(A,s) \qquad \overset{(\forall i=1\ldots n)}{\Gamma, X:A \vdash C_i:s} \qquad constructor(C_i,X)}{\Gamma \vdash \mathsf{Ind}(X:A)\{C_1|\ldots|C_n\}:A} \qquad (\mathrm{Ind}_s)$$

Example We take as an example of inductive definition, the type of lists with a given length. It is a type scheme which associate to each natural numbers n the type $(listn\ n)$ of lists of length n. The arity of this definition is $nat\to\mathsf{Set}$. There is two constructors (one for *nil* and the other one for *cons*). We want *nil* and *cons* to have type respectively $(listn\ 0)$ and $(n:nat)A\to(listn\ n)\to(listn\ (S\ n))$. We can define *listn* to be the term:

$$listn \equiv \mathsf{Ind}(X:nat\to\mathsf{Set})\{(X\ 0)|(n:nat)A\to(X\ n)\to(X\ (S\ n))\}:nat\to\mathsf{Set}$$

2.4 Constructors

New pseudo-terms correspond to the constructors (which can also be seen as introduction rules) of the inductive definition : $t := \mathsf{Constr}(i,t)$. with i a positive integer. Let I be $\mathsf{Ind}(X:A)\{C_1|\ldots|C_n\}$, the typing rule for this term is :

$$\frac{\Gamma \vdash \mathsf{Ind}(X:A)\{C_1|\ldots|C_n\}:A \qquad 1\leq i\leq n}{\Gamma \vdash \mathsf{Constr}(i,I):C_i[X\leftarrow I]} \qquad (\mathrm{Constr})$$

Example. For our example of lists we can define :

$$\begin{aligned} nil &\equiv \mathsf{Constr}(1,listn):(listn\ 0)\\ cons &\equiv \mathsf{Constr}(2,listn):(n:nat)A\to(listn\ n)\to(listn\ (S\ n))\end{aligned}$$

2.5 Eliminations

The rule for the elimination is a natural (although complicated) extension of the case of natural numbers to a general inductive definition. We shall introduce two rules corresponding to dependent and non dependent elimination.

New pseudo-terms will correspond to the eliminations of the inductive definitions : $t := \mathsf{Elim}(t,t)\{\boldsymbol{t}\}$ with $\boldsymbol{t}$ a possibly empty list of terms, written as $t_1|\ldots|t_n$. We need to define operations on forms of constructor that are used in the types of the arguments of the elimination.

Non dependent elimination Let A be an arity of sort s $(A\equiv(\boldsymbol{x}:\boldsymbol{A})s)$, let X be a variable of type A, s' be a sort, Q be a variable of type $(\boldsymbol{x}:\boldsymbol{A})s'$ and C be a type of constructor of X.

Definition 4. We define a new term $C\{X,Q\}$ by induction over the structure of the type of constructor C.

$$\begin{aligned} (P\to C)\{X,Q\} &= P\to P[X\leftarrow Q]\to C\{X,Q\} && \text{if } strict\_positive(P,X)\\ ((x:M)C)\{X,Q\} &= (x:M)C\{X,Q\} && \text{if } X \text{ does not occur in } M\\ (X\ \boldsymbol{a})\{X,Q\} &= (Q\ \boldsymbol{a}) \end{aligned}$$

Example. For our example, we have :

$$(X\ 0)\{X,Q\} = (Q\ 0)$$
$$(n\!:\!nat)A \to (X\ n) \to (X\ (S\ n))\{X,Q\} = (n\!:\!nat)A \to (X\ n) \to (Q\ n) \to (Q\ (S\ n))$$

The term $C\{X,Q\}$ is not always well-formed. It depends on the product rules allowed in the GTS. But in the case where $s = s'$, $C\{X,Q\}$ is a well-formed type. We can state more precisely that it is well-formed if "enough" products are allowed for s'.

Lemma 5. If $\Gamma, X\!:\!A \vdash C\!:\!s$ with $A \equiv (\mathbf{x} : \mathbf{A})s$ and C is a type of constructor of X. Assume s' is a sort which satisfies the following hypotheses :
1. *$(\mathbf{x} : \mathbf{A})s'$ is a well-formed type,*
2. *for all sorts s'', $(s'',s) \in \mathcal{R}$ implies $(s'',s') \in \mathcal{R}$,*
3. *if C is a recursive constructor then $(s',s') \in \mathcal{R}$,*

then the following judgment is derivable : $\Gamma, X\!:\!A, Q\!:\!(\mathbf{x} : \mathbf{A})s' \vdash C\{X,Q\}\!:\!s'$.

Let I and P be two terms of the appropriate type, we write $C\{I,P\}$ for the term $C\{X,Q\}[X \leftarrow I, Q \leftarrow P]$.

Rule Let I denotes $\mathsf{Ind}(X\!:\!A)\{C_1|\ldots|C_n\}$ with $A \equiv (\mathbf{x} : \mathbf{A})s$. Let s' be a sort. The typing rule for non-dependent elimination (s,s') is :

$$\frac{\Gamma \vdash c\!:\!(I\ \mathbf{a}) \qquad \Gamma \vdash Q\!:\!(\mathbf{x} : \mathbf{A})s' \qquad \begin{array}{c}(\forall i = 1\ldots n)\\ \Gamma \vdash f_i\!:\!C_i\{I,Q\}\end{array}}{\Gamma \vdash \mathsf{Elim}(c,Q)\{f_1|\ldots|f_n\}\!:\!(Q\ \mathbf{a})} \qquad (\mathrm{Nodep}_{s,s'})$$

Example. In our example, we get $\mathsf{Elim}(l,Q)\{x|f\}$ is well-typed of type $(Q\ n)$ if c is of type $(listn\ n)$, Q of type $nat \to s'$, x of type $(Q\ 0)$ and f of type $(n : nat)A \to (listn\ n) \to (Q\ n) \to (Q\ (S\ n))$.

Dependent elimination Let A be an arity of sort s ($A \equiv (\mathbf{x} : \mathbf{A})s$), let X be a variable of type A, let s' be a sort, let Q be a variable of type $(\mathbf{x} : \mathbf{A})(X\ \mathbf{x}) \to s'$, let C be a type of constructor of X, and c be a term of type C.

Definition 6. We define a new type $C\{X,Q,c\}$ by induction over the type of constructor C. Let P be such that *strict_positive(P, X)* ($P \equiv (\mathbf{x} : \mathbf{P})(X\ \mathbf{m})$) and M be such that X does not occur in M.

$$\begin{aligned}(P \to C)\{X,Q,c\} &= (p\!:\!P)((\mathbf{x} : \mathbf{P})(Q\ \mathbf{m}\ (p\ \mathbf{x}))) \to C\{X,Q,(c\,p)\}\\ ((x\!:\!M)C)\{X,Q,c\} &= (x\!:\!M)C\{X,Q,(c\,x)\}\\ (X\ \mathbf{a})\{X,Q,c\} &= (Q\ \mathbf{a}\ c)\end{aligned}$$

Example. For our example, with $C_2(X) \equiv (n\!:\!nat)A \to (X\ n) \to (X\ (S\ n))$

$$(X\ 0)\{X,Q,c\} = (Q\ 0\ c)$$
$$C_2(X)\{X,Q,c\} = (n\!:\!nat)(a\!:\!A)(p\!:\!(X\ n))(Q\ n\ p) \to (Q\ (S\ n)\ (c\ n\ a\ p))$$

Some restrictions are necessary to ensure that this term is well-formed. Even in the case $s' = s$, we can have troubles forming a dependent arity.

Lemma 7. Assume we have the same conditions as in lemma 5 the first one being replaced by $(\mathbf{x} : \mathbf{A})(X\ \mathbf{x}) \to s'$ is a well-formed type. The following judgment is derivable : $\Gamma, X\!:\!A, Q\!:\!(\mathbf{x} : \mathbf{A})(X\ \mathbf{x}) \to s', c\!:\!C \vdash C\{X,Q,c\}\!:\!s'$

We write $C\{I,P,t\}$ for the term $C\{X,Q,c\}[X \leftarrow I, Q \leftarrow P, c \leftarrow t]$.

Rule Let I denotes $\mathsf{Ind}(X:A)\{C_1|\ldots|C_n\}$ with $A \equiv (x:A)s$. Let s' be a sort. The typing rule for dependent elimination (s,s') is :

$$\frac{\Gamma \vdash c:(I\ a) \quad \Gamma \vdash Q:(x:A)(I\ x)\rightarrow s' \quad \overset{(\forall i = 1\ldots n)}{\Gamma \vdash f_i:C_i\{I,Q,\mathsf{Constr}(i,I)\}}}{\Gamma \vdash \mathsf{Elim}(c,Q)\{f_1|\ldots|f_n\}:(Q\ a\ c)} \quad (\mathrm{Dep}_{s,s'})$$

Example. In our example, we get $\mathsf{Elim}(l,Q)\{x|f\}$ is well-typed of type $(Q\ n\ l)$ if c is of type $(\mathit{listn}\ n)$, Q of type $(n:\mathit{nat})(\mathit{listn}\ n)\rightarrow s'$, x of type $(Q\ 0\ \mathit{nil})$ and f of type $(n:\mathit{nat})(a:A)(m:(\mathit{listn}\ n))(Q\ n\ m)\rightarrow(Q\ (S\ n)\ (\mathit{cons}\ n\ a\ m))$.

Discussion Obviously the non-dependent elimination can be coded using a dependent elimination over a constant predicate. We prefer to distinguish the two operations, because in some systems, the term $(x:A)(I\ x)\rightarrow s$ is not well-formed, although $(x:A)s$ is (note that the rule $\mathrm{Nodep}_{s,s}$ can be added in any GTS). We use the same syntax for both constructions. It is not problematic because first, the two eliminations behaves the same w.r.t. the reduction rule, second we can solve the ambiguity using the type information. More precisely if the system is normalizing, given an environment Γ and a term t, we can find if there exists a term M such that $\Gamma \vdash t : M$. If t is $\mathsf{Elim}(c,Q)\{f\}$ then t is well-typed if and only if :
- c is well typed of type M.
- M is reducible to $(I\ a)$ with $I \equiv \mathsf{Ind}(X:A)\{C_1|\ldots|C_n\}$ and $A \equiv (x:A)s$.
- f has length n (we write it as $f_1|\ldots|f_n$).
- Q is well-typed of type an arity A'.
- Either A' is convertible with $(x:A)s'$, $\mathrm{Nodep}_{s,s'}$ is an allowed rule and for each i, f_i is well-typed in Γ and its type is convertible to $C_i\{I,Q\}$. In that case the type of t is $(Q\ a)$. Or A' is convertible with $(x:A)(I\ x)\rightarrow s'$ and $\mathrm{Dep}_{s,s'}$ is an allowed rule and for each i, f_i is well-typed in Γ and its type is convertible to $C_i\{I,Q,\mathsf{Constr}(i,I)\}$. In that case the type of t is $(Q\ a\ c)$.

2.6 Rules for conversion

The rules for conversion are extended w.r.t. the new operations for defining terms.

$$\frac{A \simeq A' \qquad \overset{(\forall i = 1\ldots n)}{C_i \simeq C'_i}}{\mathsf{Ind}(X:A)\{C_1|\ldots|C_n\} \simeq \mathsf{Ind}(X:A')\{C'_1|\ldots|C'_n\}}$$

$$\frac{M \simeq M' \qquad i = i'}{\mathsf{Constr}(i,M) \simeq \mathsf{Constr}(i',M')} \qquad \frac{c \simeq c' \qquad P \simeq P' \qquad \overset{(\forall i = 1\ldots n)}{f_i \simeq f'_i}}{\mathsf{Elim}(c,P)\{f_1|\ldots|f_n\} \simeq \mathsf{Elim}(c',P')\{f'_1|\ldots|f'_n\}}$$

A new reduction rule (called ι-reduction) is added. We need some notations to state it. Let I denotes $\mathsf{Ind}(X:A)\{C_1|\ldots|C_n\}$ with $A \equiv (x:A)s$. Let F and f be two terms, let X be a variable of type A and C be a type of constructor of X.

Definition 8. We define a new term $C[X,F,f]$ by induction over C which is a type of constructor of X. Let $P \equiv (x:P)(X\ m)$ be such that *strict_positive(P, X)* and M such that X does not occur in M.

$$\begin{array}{ll} (P \rightarrow C)[X,F,f] & = [p:P]C[X,F,(f\ p\ [x:P](F\ m\ (p\ x)))] \\ ((x:M)C)[X,F,f] & = [x:M]C[X,F,(f\ x)] \\ (X\ a)[X,F,f] & = f \end{array}$$

Example. For our example, we have :

$$(X\ 0)[X,F,f] = (f\ 0)$$
$$(n:nat)A \rightarrow (X\ n) \rightarrow (X\ (S\ n))[X,F,f] = [n:nat][a:A][p:(X\ n)](f\ n\ a\ p\ (F\ n\ p))$$

$C[X,F,f]$ is well-typed both in the dependent and the non dependent case.

Lemma 9. Let s' be a sort, C be a type of constructor of X and $C \equiv (z:C)(X\ a)$.

- *Let Δ be $\Gamma, X:A, Q:(x:A)s', F:(x:A)(X\ x) \rightarrow (Q\ x)$. If $C\{X,Q\}$ is a well-formed type in Δ, then the judgment $\Delta, f:C\{X,Q\} \vdash C[X,F,f]:(z:C)(Q\ a)$ is provable.*
- *Let Δ be $\Gamma, X:A, Q:(x:A)(X\ x) \rightarrow s', F:(x:A)(c:(X\ x))(Q\ x\ c)$. If $C\{X,Q,c\}$ is a well-formed type in $\Delta, c:C$ then the judgment $\Delta, c:C, f:C\{X,Q,c\} \vdash C[X,F,f]:(z:C)(Q\ a\ (c\ z))$ is provable.*

As usual $C[I,G,g]$ will denote $C[X,F,f]\ [X \leftarrow I, F \leftarrow G, f \leftarrow g]$.

The ι reduction. Let s' be a sort, Q be a term and f be a sequence of n terms. We define : $\mathsf{Fun\_Elim}(I,Q,f) = [x:A][c:(I\ x)]\mathsf{Elim}(c,Q)\{f\}$.
The reduction rule is, if f_i denotes the $i-th$ element of the sequence f:

$$\mathsf{Elim}((\mathsf{Constr}(i,I)\ m),Q)\{f\} \longrightarrow_\iota (C_i[I,\mathsf{Fun\_Elim}(I,Q,f),f_i]\ m) \qquad (\iota\text{-red})$$

Example. For our example, we have :

$$\begin{array}{ll} \mathsf{Fun\_Elim}(listn,Q,x|f) & = [n:nat][l:(listn\ n)]\mathsf{Elim}(l,Q)\{x|f\} \\ \mathsf{Elim}(nil,Q)\{x|f\} & \longrightarrow_\iota x \\ \mathsf{Elim}((cons\ n\ a\ p),Q)\{x|f\} & \longrightarrow_\iota \\ \multicolumn{2}{l}{([n:nat][a:A][p:(listn\ n)](f\ n\ a\ p\ (\mathsf{Fun\_Elim}(listn,Q,x|f)\ n\ a\ p)}\\ & \longrightarrow_\beta^* (f\ n\ a\ p\ \mathsf{Elim}(p,Q)\{x|f\}) \end{array}$$

3 The system Coq : definition

3.1 The underlying GTS

The system Coq has two sorts (Set and Prop) at the impredicative level. Set and Prop distinguish between proofs that are interpreted as programs and proofs that are only a justification of some logical part.

The underlying GTS is built from $\mathcal{S} = \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}, \mathsf{Type}_S\}$ with axioms and $\mathcal{A} = \{\mathsf{Prop}:\mathsf{Type}, \mathsf{Set}:\mathsf{Type}_S\}$ and rules $\mathcal{R} = \mathcal{S} \times \mathcal{S}$.

3.2 Rules for inductive definitions

Inductive definitions can be defined both for Prop and Set, so we have the rules $\mathrm{Ind}_{\mathsf{Prop}}$ and $\mathrm{Ind}_{\mathsf{Set}}$.

Weak eliminations In Coq, the rules for elimination are : $\mathrm{Dep}_{\mathsf{Set},\mathsf{Set}}$, $\mathrm{Dep}_{\mathsf{Set},\mathsf{Prop}}$, $\mathrm{Nodep}_{\mathsf{Prop},\mathsf{Prop}}$.
These rules make sense because we may build the same products over Set and Prop.

We do not allow $\mathrm{Nodep}_{\mathsf{Prop},\mathsf{Set}}$ because, in general, we cannot build a program by case analysis over the structure of a proof that is discarded for computation.

There is no strong reason to avoid the rule $\mathrm{Dep}_{\mathsf{Prop},\mathsf{Prop}}$. But our interpretation of $\Gamma \vdash a : A$ with A : Prop is that A is provable, so we are not interested in a as an object to reason about.

Strong eliminations. With the system above we shall not be able to prove for instance that 0 is not equal to 1. For this, one possibility is to allow a strong elimination, namely a rule like $\mathrm{Nodep}_{\mathsf{Set},\mathsf{Type}}$. This rule is allowed in Coq but only for a restricted class of inductive definitions that are called *small* inductive definitions.

Definition 10. A form of constructor of X is said to be *small* if it is $(X\ a)$ or $(x : M)C$ with C a small constructor of X and M a "small type" ie of type Prop or Set (and not Type or Type_S).

An inductive type $\mathsf{Ind}(X : A)\{C_1 | \ldots | C_n\}$ is said to be small if all the forms of constructor C_i are small.

In Coq the strong elimination scheme $\mathrm{Nodep}_{\mathsf{Set},\mathsf{Type}}$ is only allowed for small inductive types. The rule can be written as :

let $I = \mathsf{Ind}(X : (x : A)\mathsf{Set})\{C_1 | \ldots | C_n\}$ be a small inductive type :

$$\frac{\Gamma \vdash c : (I\ a) \qquad \Gamma \vdash Q : (x : A)\mathsf{Type} \qquad \begin{array}{c}(\forall i = 1 \ldots n)\\ \Gamma \vdash f_i : C_i\{I, Q\}\end{array}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1 | \ldots | f_n\} : (Q\ a)} \quad (\mathrm{Nodep}_{\mathsf{Set},\mathsf{Type}})$$

A problem to state this rule is to be able to write the type of Q in the system. For instance in the pure Calculus of Constructions, it is only possible to write Q : Type with A an empty list. Also we cannot express the corresponding dependent elimination. But in the Coq system, there is a hierarchy of universes, so we can build arities on Type and there is no problem in the expression of the elimination scheme.

Other possible strong eliminations. We cannot allow strong eliminations for non-small inductive types without getting an inconsistency. This can be shown with an adaptation of the argument developed in [6] to this system.

The point is that we can always build an inductive type B : Set with one constructor $\varepsilon : \mathsf{Prop} \rightarrow B$. With the rule $\mathrm{Nodep}_{\mathsf{Set},\mathsf{Type}}$ we can build a projection $E : B \rightarrow \mathsf{Prop}$ and we will have $(E\ (\varepsilon\ A))$ and A convertible for each A of type Prop. So we get an object of type Set which is isomorphic to Prop. Because we have impredicativity at both levels, it is not surprising we get an inconsistency.

We do not allow the rule $\text{Nodep}_{\text{Prop},\text{Type}}$ even for small inductive definitions. Actually if such a rule was available, we could find $A:\text{Prop}$, $a,b:A$ and prove $a \neq b$. This goes against the intended interpretation of the Prop part of the system with a proof-irrelevance semantics (all proofs of a proposition are identified). However this rule forbids the use of other natural principle for instance the axiom for extensionality $(A,B:\text{Prop})(A \rightarrow B) \rightarrow (B \rightarrow A) \rightarrow A = B$ or the excluded middle $(A:Prop)A \vee \neg A$ as shown by Berardi or Coquand [6].

We do not allow the rule $\text{Nodep}_{\text{Set},\text{Type}_S}$ although it could have very interesting applications. But this rule destroys the interpretation of proofs in Coq as non-dependent programs of F_ω extended with inductive definitions. We could for instance build a type $T(n)$ such that $T(\tilde{n})$ is the n-ary product of a type A. This is really a non-trivial extension of the system from the computational point of view.

B. Werner [23] recently proved the normalization property for a second-order polymorphic system extended with a type of natural numbers together with weak and strong eliminations. If the proof can be extended to the full system, it will be possible to introduce an extended Coq system with the rule $\text{Nodep}_{\text{Set},\text{Type}_S}$.

Inductive types and universes. The system Coq is build on a GTS which extends the Calculus of Constructions with an implicit hierarchy of universes. An alternative extension of the Calculus is to introduce inductive types at the predicative level. If we want to be exact, we have to say that, in the system Coq, the rules Ind_{Type}, $\text{Dep}_{\text{Type},\text{Prop}}$, and $\text{Dep}_{\text{Type},\text{Type}}$ are possible. But they were never used in the examples developed so far. Also the interaction between the implicit hierarchy of universes and the elimination scheme was not carefully checked.

3.3 Examples

We give a few examples of "typical" rules to give a flavor of what we get. We shall state the introduction rules (the types of the constructor), the elimination scheme (actually the type of the "generic" elimination scheme $[Q][f]\text{Fun\_Elim}(I,Q,f)$ either in the dependent or the nondependent case) and the reduction rules.

Absurd proposition We can define a proposition $\bot \equiv \text{Ind}(X:\text{Prop})\{\}$ without constructors. The non-dependent elimination will have type : $(P:\text{Prop})\bot \rightarrow P$ which says that from absurdity you can prove any proposition.

Product type Let A and B be two types. The product of A and B will be defined as a type $A * B$ with only one constructor *pair* of type $A \rightarrow B \rightarrow A * B$.

$$A * B \equiv \text{Ind}(X:\text{Set})\{A \rightarrow B \rightarrow X\}$$

The dependent elimination has type :

$$prod\_rec : (P:A * B \rightarrow \text{Set})((a:A)(b:B)(P\ (pair\ a\ b))) \rightarrow (x:A * B)(P\ x)$$

which says that any element in the type $A * B$ is equivalent to $(pair\ a\ b)$ for some a in A and b in B. The non dependent elimination of $A * B$ on a type C has type $(A \rightarrow B \rightarrow C) \rightarrow A * B \rightarrow C$ which corresponds to the uncurryfication. The rule for conversion is : $(prod\_rec\ P\ f\ (pair\ a\ b)) \simeq (f\ a\ b)$

Sum type The disjoint union $(A+B)$ of A and B has two constructors *inl* of type $A \rightarrow (A+B)$ and *inr* of type $B \rightarrow (A+B)$. $A+B \equiv \mathsf{Ind}(X:\mathsf{Set})\{A \rightarrow X | B \rightarrow X\}$.
The dependent elimination has type :
$sum\_rec : (P:A+B \rightarrow \mathsf{Set})((a:A)(P\ (inl\ a))) \rightarrow ((b:B)(P\ (inr\ b))) \rightarrow (x:A+B)(P\ x)$

The rules for conversion are :

$$(sum\_rec\ P\ f\ g\ (inl\ a)) \simeq (f\ a) \qquad (sum\_rec\ P\ f\ g\ (inr\ b)) \simeq (g\ b)$$

Remark that the constructors and eliminations for disjoint union or product are polymorphic with respect to the types A and B.

Equality An example of an inductively defined predicate is the equality. It is a predicate $eq_{A,x}$ parameterized by a set A and an element x of A. We say that the set of elements of A equal to x is the smallest set which contains x. The arity of $eq_{A,x}$ is $A \rightarrow \mathsf{Prop}$, it has one constructor *refl_equal* of type $(eq_{A,x}\ x)$.

$$eq_{A,x} \equiv \mathsf{Ind}(X:A \rightarrow \mathsf{Prop})\{(X\ x)\}$$

The non-dependent elimination has type :

$$eq_{A,x}\text{-}ind : (P:A \rightarrow \mathsf{Prop})(P\ x) \rightarrow (y:A)(eq_{A,x}\ y) \rightarrow (P\ y)$$

This principle says that if x and y are equal, to prove $(P\ y)$ it is enough to prove $(P\ x)$.

Well-founded induction Assume we have a set A and an arbitrary relation $R:A \rightarrow A \rightarrow \mathsf{Prop}$. We define the set of elements of A which are accessible for the relation R as the smallest set which contains x if it contains all its predecessors for R. The accessibility set *Acc* has arity $A \rightarrow \mathsf{Prop}$. The introduction rule *Acc_intro* has type : $(x:A)((y:A)(R\ y\ x) \rightarrow (Acc\ y)) \rightarrow (Acc\ x)$

$$Acc \equiv \mathsf{Ind}(X:A \rightarrow \mathsf{Prop})\{(x:A)((y:A)(R\ y\ x) \rightarrow (X\ y)) \rightarrow (X\ x)\}$$

The non dependent elimination principle is :

$$\begin{aligned} Acc\_ind: & (P:A \rightarrow \mathsf{Prop}) \\ & ((x:A)((y:A)(R\ y\ x) \rightarrow (Acc\ y)) \rightarrow ((y:A)(R\ y\ x) \rightarrow (P\ y)) \rightarrow (P\ x)) \\ & \rightarrow (x:A)(Acc\ x) \rightarrow (P\ x) \end{aligned}$$

3.4 Extraction of programs

It is possible to define a realisability interpretation for the system Coq, that will extract programs in the system F_ω plus inductive definitions (namely the system F_ω plus the rules Ind_{Set}, Constr and $\text{Nodep}_{\text{Set,Set}}$).

As usual the informative contents of a term corresponds to the sort of its type. We remark that the informative contents of $\mathsf{Elim}(t,Q)\{f\}$ is the one of Q and consequently f and if it is informative then t also is informative (this property would not be true if $\text{Nodep}_{\text{Prop,Set}}$ was a rule).

The extension of the extraction function defined in [20, 21] for informative terms involving inductive definitions is the following :

$$\begin{aligned} &\mathcal{E}(\mathsf{Ind}(X:A)\{C_1|\ldots|C_n\}) = \mathsf{Ind}(X:\mathcal{E}(A))\{\mathcal{E}(C_1)|\ldots|\mathcal{E}(C_n)\} \\ &\mathcal{E}(\mathsf{Constr}(i,ind)) \qquad\quad = \mathsf{Constr}(i,\mathcal{E}(ind)) \\ &\mathcal{E}(\mathsf{Elim}(t,Q)\{f\}) \qquad\ \ = \mathsf{Elim}(\mathcal{E}(t),\mathcal{E}(Q))\{\mathcal{E}(f)\} \end{aligned}$$

It is easy to check that the extraction function preserves typability.

4 Meta-theory

4.1 The Calculus of Constructions with fixpoints

In order to study the meta-theory of the system Coq, we shall use a different extension of the Calculus of Constructions introduced by Ph. Audebaud [1, 2]. His idea was to extend the Calculus of Constructions with a fixpoint in order to encode efficient inductive types and also to reason about partial objects. A fixpoint operator is introduced at the level of both types and programs. For types it is assumed that it only applies to positive operators, in a sense that will be explained after.

We do not need the full power of this calculus and in particular we are not interested in partial objects. We will only consider a subsystem of CC+ that we call CC_μ. This system is the pure Calculus of Constructions with a fixpoint operator for positive types transformers. We shall also need only one sort that we also call Set.

Definitions The terms contain the pure terms of the Calculus of Constructions plus a term for fixpoint formation that will be written $<x:t>t'$.

The set of *positive* and *negative* terms with respect to a variable X are defined by the following grammar :

$$\begin{aligned} Pos &::= M \mid X \mid (Pos\ m) \mid [x:M]Pos \mid (x:Neg)Pos \\ Neg &::= M \mid (Neg\ m) \mid [x:M]Neg \mid (x:Pos)Neg \end{aligned}$$

with the restriction that X does not occur in M or m. If M is strictly positive with respect to X then it is also positive with respect to X. We shall write *positive(P, X)* to indicate that P is positive with respect to X.

Rules The rules for this calculus are the ones for a GTS with axioms : $\mathcal{A} = \{\mathsf{Set} : \mathsf{Type}_{\mathcal{S}}\}$ and $\mathcal{S} \times \mathcal{S}$ as the set of rules. Furthermore the conversion rules are extended with the reduction step for fixpoints :

$$<x:A>M \longrightarrow ([x:A]M\ <x:A>M) \qquad \text{(fix-red)}$$

There is one new rule for fixpoint introduction at the level of types :

$$\frac{\Gamma \vdash A : \mathsf{Type}_{\mathcal{S}} \qquad \Gamma, X:A \vdash M:A \qquad positive(M,X)}{\Gamma \vdash <X:A>M : A} \qquad (\text{fixintro}_{\mathsf{Type}_{\mathcal{S}}})$$

Properties Ph. Audebaud proved that CC+ enjoys the Church-Rosser property and also strong normalization for β-reduction (of course not for fix-red) We can embed CC_μ in CC+ just transforming Set and $\mathsf{Type}_{\mathcal{S}}$ into the *Prop* and *Type* sorts of CC+. Consequently, we have strong normalization for β-reduction in CC_μ as well.

4.2 The system F_ω^{Ind}

We introduce a subsystem F_ω^{Ind} of Coq which is a F_ω plus inductive definitions extended with weak non dependent eliminations. We define a transformation from F_ω^{Ind} to the system CC_μ that gives the strong normalization result for β and ι reductions in F_ω^{Ind}.

The system F_ω^{Ind} is the inductive GTS with sorts $\mathcal{S} = \{\mathsf{Set}, \mathsf{Type}_\mathcal{S}\}$, axioms $\mathcal{A} = \{\mathsf{Set} : \mathsf{Type}_\mathcal{S}\}$, rules $\mathcal{R} = \{(\mathsf{Set},\mathsf{Set}); (\mathsf{Type}_\mathcal{S}, \mathsf{Set}); (\mathsf{Type}_\mathcal{S}, \mathsf{Type}_\mathcal{S})\}$ and with inductive definitions of sort Set and only non-dependent elimination of sort (Set,Set). The added rules are $\mathrm{Ind}_{\mathsf{Set}}$, Constr and $\mathrm{Nodep}_{\mathsf{Set},\mathsf{Set}}$.

4.3 Translation from F_ω^{Ind} to CC_μ

We define a translation from terms in the Calculus of Constructions into terms in CC_μ. $\overline{M}$ will denote the translation of the term M and if Γ is the environment $x_1 : M_1, \ldots, x_n : M_n$ of CC_μ, we write $\overline{\Gamma}$ the environment $x_1 : \overline{M_1}, \ldots, x_n : \overline{M_n}$ of CC_μ.

We can prove that $\Gamma \vdash_{coq} M : N$, implies $\overline{\Gamma} \vdash_{CC_\mu} \overline{M} : \overline{N}$. This is done by induction over the proof of $\Gamma \vdash_{coq} M : N$. We do not give here the justifications of the correctness of this translation, full details could be find in an extended version of this paper.

Translating the pure part The systems F_ω^{Ind} and CC_μ have a common part (terms and rules) corresponding to F_ω. The translation is defined as a structural morphism on this part.

$$\begin{aligned}
\overline{\mathsf{Set}} &= \mathsf{Set} \\
\overline{x} &= x \quad \text{if } x \text{ is a variable} \\
\overline{(x:A)B} &= (x:\overline{A})\overline{B} \\
\overline{[x:A]B} &= [x:\overline{A}]\overline{B} \\
\overline{(A\ B)} &= (\overline{A}\ \overline{B})
\end{aligned}$$

The interesting cases of the translation correspond to the terms involving inductive definitions.

Translating the inductive definitions In the system CC_μ, using fixpoints, we can identify the definition of the type and the expected recursive scheme.

For the natural numbers, it will give us the following representation $\overline{nat} \equiv \langle X:*\rangle(C:*)C \to (X \to C \to C) \to C$. The recursor is just the identity function. A similar representation was proposed in the framework of AF_2 by M. Parigot [19]. The main drawback of this encoding of natural numbers is that the representation of the nth natural number uses a space proportional to 2^n. This is a problem for practical uses but not for our study of the theoretical properties of the system Coq.

The general case. Let I be $\mathsf{Ind}(X : A)\{C_1|\ldots|C_n\}$ and $A \equiv (\boldsymbol{x} : \boldsymbol{A})\mathsf{Set}$. We define

$$\overline{I} = <X:\overline{A}>[\boldsymbol{x} : \overline{\boldsymbol{A}}](Q:\overline{A})\overline{C_1}\{X,Q\} \rightarrow \cdots \rightarrow \overline{C_n}\{X,Q\} \rightarrow (Q\ \boldsymbol{x})$$

With this definition the translation of the elimination is almost trivial :

$$\overline{\mathsf{Elim}(c,Q)\{\boldsymbol{f}\}} = (\overline{c}\ \overline{Q}\ \overline{\boldsymbol{f}})$$

$\overline{\mathsf{Fun\_Elim}(I,Q,\boldsymbol{f})} = [\boldsymbol{x}:\overline{A}][c:(\overline{I}\ \boldsymbol{x})](c\ \overline{Q}\ \overline{\boldsymbol{f}})$ which has type $(\boldsymbol{x}:\overline{A})(\overline{I}\ \boldsymbol{x}) \rightarrow (\overline{Q}\ \boldsymbol{x})$.

The translation of constructors is as follows. We assume $I = \mathsf{Ind}(X : A)\{C_1|\ldots|C_n\}$, i such that $1 \leq i \leq n$ and $C_i \equiv (\boldsymbol{z} : \boldsymbol{C})(X\ \boldsymbol{a})$. We expect $\overline{\mathsf{Constr}(i,I)}$ to be of type $\overline{C_i[X \leftarrow I]}$ ie $(\boldsymbol{z}:\overline{\boldsymbol{C}[X \leftarrow I]})(\overline{I}\ \overline{\boldsymbol{a}})$. We take with $\boldsymbol{f} = f_1|\ldots|f_n$:

$$\overline{\mathsf{Constr}(i,I)} = [\boldsymbol{z}:\overline{\boldsymbol{C}[X \leftarrow I]}][Q:\overline{A}][f_1:\overline{C_1}\{\overline{I},Q\}]\ldots[f_n:\overline{C_n}\{\overline{I},Q\}]$$
$$(\overline{C_i}[\overline{I}, \overline{\mathsf{Fun\_Elim}(I,Q,\boldsymbol{f})}, f_i]\ \boldsymbol{z})$$

We have to show that if $M \equiv_{coq} N$ then $\overline{M} \equiv_{CC_\mu} \overline{N}$. The β-reduction rule does not lead to any problem. For the inductive reduction, it is easy to check that

$$\overline{\mathsf{Elim}((\mathsf{Constr}(i,I)\ \boldsymbol{m}),Q)\{\boldsymbol{f}\}} \longrightarrow^{+}_{\beta} \overline{(C_i[I, \mathsf{Fun\_Elim}(I,Q,\boldsymbol{f}), f_i]\ \boldsymbol{m})}$$

So we have if M reduces to N in one step of inductive reduction then $\overline{M}$ reduces to $\overline{N}$ in at least one step of β-reduction. Remark that we do not use fix-reduction at this stage, this rule is only used to make sure that the elimination and constructors are well-typed.

We can deduce from these results that there cannot be infinite sequences of ι and β reduction in $F_\omega^{\mathrm{Ind}}$and consequently :

Theorem 11. *The system F_ω^{Ind} is strongly normalizing.*

The same translation shows the strong normalization of the extension of pure Calculus of Constructions with Inductive definitions and weak elimination.

4.4 The Calculus of Inductive Definitions

From the previous study we can deduce properties of the full Calculus of Inductive Definitions as defined in section 3.

Weak elimination The distinction between Prop and Set does not matter for the study of the normalization properties. We can just map Prop and Type to respectively Set and Type$_S$. First let us consider the Calculus of Constructions with only weak (possibly dependent) eliminations. We can map this calculus (called Coq_w) into the system F_ω^{Ind} using a transformation which forgets dependencies. This is an extension of the map which transforms a term of the Calculus of Constructions into a term of F_ω as described for instance in [20, 21]. It is easy to show that as the Calculus of Constructions is conservative over F_ω, the system Coq_w is conservative over F_ω^{Ind}. The normalization for F_ω^{Ind} implies its consistency (there can be no closed normal proof of $(C:\mathrm{Set})C$) and then the consistency of Coq_w. The translation from Coq_w to F_ω^{Ind} preserves the underlying pure lambda terms (without any type information). We believe (but did not check precisely the details) that the method of translation from the pure Calculus of Constructions to F_ω used by Geuvers and Nederhof [13] could also be adapted to Coq_w and F_ω^{Ind} in order to justify strong normalization for Coq_w.

Strong elimination To deal with strong elimination is more complicated because in a system with strong elimination we cannot anymore ignore dependencies with respect to programs. Also we know that a careless use of strong elimination leads to paradoxes.

The scheme for strong elimination in Coq is restricted to the rule $\text{Nodeps}_{\text{Set},\text{Type}}$. This rule is sufficient to prove for instance $\neg(true = false)$.

What we actually use of this rule in the examples developed so far could be obtained by just adding the axiom $\neg(true = false)$ to Coq_w and not the full scheme of strong elimination. If A:Type in Coq_w with only weak elimination, then $A \equiv (x : A)\text{Prop}$. Now with $\text{Nodeps}_{\text{Set},\text{Type}}$ we can build a function of type $nat \rightarrow A$ by giving F of type A and G of type $nat \rightarrow A \rightarrow A$. Furthermore, we shall have $(H\ 0)$ convertible with F and $(H\ (S\ n))$ convertible with $(G\ n\ (H\ n))$.

Using only $\neg(true = false)$ and weak eliminations we could internally represent a term H of type $nat \rightarrow A$ such that $(H\ 0) \Leftrightarrow_A F$ and $(H\ (S\ n)) \Leftrightarrow_A (G\ n\ (H\ n))$. The equivalence $\Leftrightarrow_A$ being defined as $[p, q : A](x : A)((p\ x) \rightarrow (q\ x) \wedge (q\ x) \rightarrow (p\ x))$. Obviously the proofs are shorter to do with the strong elimination scheme than with the internal encoding.

5 Discussion

5.1 Allowing more positive types

One restriction in our proposition for an extension with inductive definition is in the shape required for a form of constructors and mainly the condition of strict positivity.

We could relax this condition in two ways. The first one is to ask only for a positivity condition (as defined in section 4.1). As it is explained in [8], with only a positivity condition and if we allow the rule Ind_{Type} and the elimination $\text{Nodep}_{\text{Type},\text{Type}}$, we get a paradox. But in the system without the rule Ind_{Type}, such an extension could be justified by a translation in CC_μ.

We could also extend the definition of strictly positive by saying X is strictly positive in $\text{Ind}(Y : A)\{C_1 | \ldots | C_n\}$ if X occurs strictly negatively in each C_i (we say that X occurs strictly negatively in $(z : C)(X\ a)$ if X does not occur or occurs strictly positively in each term of C). This possibility follows the general habit to say that X is strictly positive in $A + B$ or $A * B$ if it is strictly positive or does not occur in A and B. This allows also to define mutually inductive types by an encoding using several levels of inductive definitions.

But the first question with a weaker notion of positivity is how to define naturally the elimination principles, namely the auxiliary functions $C\{X, P\}$ or $C\{X, P, c\}$. There are several possibilities either using a product (for the non-dependent case) or a strong sum (for the dependent case) together with projection (it works for the two extensions of the positivity condition). In the case of an extension with several levels of inductive definitions, we can use a strong elimination scheme in order to define the predicates $C\{X, P\}$ and $C\{X, P, c\}$ but it is not really satisfactory to use a so strong scheme for defining the type of even a weak elimination. Another possibility is the one proposed by Mendler [16] which can be extended to the dependent elimination in our formalism and does not involve auxiliary operations. But none of this possibilities give a really natural formulation of the elimination principle. We think that the most natural solution for mutually inductive definitions will be to introduce them as a primitive notion by a simple generalization of the rules presented in this paper.

5.2 Elimination for inductive predicate

As mentioned by Th. Coquand in [7], the general elimination scheme proposed in this paper is not fully adequate for the case of inductive predicate, because it does not take into account the fact that, for instance, an object in the type $(listn\ (S\ n))$ can only be built from the second constructor. He proposed an alternative presentation of the elimination scheme similar to a definition by pattern-matching in functional languages. With this method we can easily prove $\neg(true = false)$ without introducing the full power of strong elimination.

5.3 Conclusion

The main ideas of the formulation of the inductive definitions were already in [8] and a similar proposition was simultaneously done by [9]. The purpose of this paper was to give the precise definitions corresponding to the system Coq and also to show that strong normalization for a large useful subsystem of Coq could be obtained from previous results known about the Calculus of Constructions.

The main point about this presentation is that it corresponds to a closed system. This is different from the extension with inductive definitions provided in other systems like Lego [14] or Alf [15] also based on typed lambda-calculus. In these systems new constants and new equality rules can be added in the system. It is the user responsibility to check that adding these new objects and rules is safe.

Our proposition was intended to be of practical use and integrated to the previous implementation of the Calculus of Constructions. Indeed, with these operations, the type recognition procedure or the discharge operation were easy to extend. Also having only one kind of constants (as abbreviation) makes a lot of things simpler. It is less clear whether this presentation is well-suited for proof synthesis. For instance, defining the addition function going back to the elimination scheme is a bit delicate and surely users will prefer to define such a function using equalities or an ML-like syntax. Also assume that addition is defined for instance as :

$$add \equiv [n, m : nat]\mathsf{Elim}(n, nat)\{m, [p, addmp : nat](S\ addmp)\}$$

Then we will have $(add\ (S\ n)\ m)$ is convertible with $(S\ (add\ n\ m))$ but we do not have $(add\ (S\ n)\ m)$ reduces to $(S\ (add\ n\ m))$. Obviously we would like such an equality corresponding to the intended definition of the addition to be automatically recognized by the system. In conclusion, our opinion is that the status of names involved in the inductive definitions has still to be clearly understood.

Acknowledgments

We thanks G. Huet, B. Werner and A. Sellink for comments on a previous version of the paper.

References

1. Ph. Audebaud. Partial objects in the calculus of constructions. In *Proceedings of the sixth Conf. on Logic in Computer Science.* IEEE, 1991.
2. Ph. Audebaud. *Extension du Calcul des Constructions par Points fixes.* PhD thesis, Université Bordeaux I, 1992.
3. H. Barendregt. Lambda calculi with types. Technical Report 91-19, Catholic University Nijmegen, 1991. in Handbook of Logic in Computer Science, Vol II.
4. C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.
5. R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.
6. Th. Coquand. Metamathematical investigations of a Calculus of Constructions. In P. Oddifredi, editor, *Logic and Computer Science.* Academic Press, 1990. Rapport de recherche INRIA 1088, also in [11].
7. Th. Coquand. Pattern matching with dependent types. In Nordström et al. [17].
8. Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88.* Springer-Verlag, 1990. LNCS 417.
9. P. Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59-79, 1990.
10. G. Dowek et al. The Coq Proof Assistant User's Guide Version 5.6. Rapport Technique 134, INRIA, December 1991.
11. G. Huet ed. *The Calculus of Constructions, Documentation and user's guide, Version V4.10*, 1989. Rapport technique INRIA 110.
12. H. Geuvers. Type systems for Higher Order Logic. Faculty of Mathematics and Informatics, Catholic University Nijmegen, 1990.
13. H. Geuvers and M.-J. Nederhof. A modular proof of strong normalization for the Calculus of Constructions. Faculty of Mathematics and Informatics, Catholic University Nijmegen, 1989.
14. Z. Luo and R. Pollack. Lego proof development syste : User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh., 1992.
15. L. Magnusson. The new implementation of ALF. In Nordström et al. [17].
16. N. Mendler. Recursive types and type constraints in second order lambda-calculus. In *Symposium on Logic in Computer Science*, Ithaca, NY, 1987. IEEE.
17. B. Nordström, K. Petersson, and G. Plotkin, editors. *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.
18. P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory. Bibliopolis, 1984.
19. M. Parigot. On the representation of data in lambda-calculus. In *CSL'89*, volume 440 of *LNCS*, Kaiserslautern, 1989. Springer-Verlag.
20. C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.
21. C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Université Paris 7, January 1989.
22. F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, LNCS 442. Springer-Verlag, 1990. also technical report CMU-CS-89-209.
23. B. Werner. A normalization proof for an impredicative type system with large elimination over integers. In Nordström et al. [17].