Primitive Recursion for Higher Order Abstract Syntax with Dependent Types

Joëlle Despeyroux and Pierre Leleu *

INRIA

Abstract. Higher Order Abstract Syntax (HOAS) and induction are two central representation techniques in logical frameworks. Unfortunately, a type defined by HOAS is not an inductive type in usual systems such as the Calculus of Inductive Constructions. There have been recent systems proposals with primitive recursive operators over types defined through HOAS ([DPS97,Lel97,DL98]). These systems are based on different presentations of modal cores IS4 with simple types.
In this article, we present an extension of the system [DL98] to dependent types. The new system is still based on a modal core and has iteration and case analysis operators. However, dependent types create new problems that we solve by putting restrictions on the terms of the system on which we are allowed to iterate. Proving properties of this system is a hard task, we have only established the properties of confluence and type preservation by reduction for the moment. We claim this system to represent the most achieved step towards total reconciliation between HOAS and induction so far.

1 Introduction

When representing a formal system in a typed theory, induction and Higher Order Abstract Syntax (HOAS) prove to be two central tools. HOAS is a representation technique which greatly eases modelling variable bindings. Its basic idea is to represent object variables by meta variables. For example, to represent the type *term* of the untyped λ-terms with HOAS, we simply define the following two constructors, one for the application and one for the abstraction:

$$\begin{array}{l}\mathrm{app} : \mathrm{term} \to \mathrm{term} \to \mathrm{term} \\ \mathrm{lam} : (\mathrm{term} \to \mathrm{term}) \to \mathrm{term}\end{array}$$

The type of constructor *lam* shows that the object variable binding is represented by a meta variable binding. Because of the negative (first) occurrence of type term, term cannot be defined as an inductive type in usual inductive type theories (such as the Calculus of Inductive Constructions).

In this work we concentrate on recovering recursion on types such as term defined in HOAS. In the cases when the meta-theory has only simple types, some solution has already been proposed. Joëlle Despeyroux, Frank Pfenning and Carsten Schürmann have presented a system based on a modal λ-calculus IS4 with two recursive operators over HOAS types: one for case analysis *Case* and another one for iteration *It* ([DPS96,DPS97]).

What is the principle of this system? The problems we have to solve when considering recursion over HOAS types are twofold:

- First, we have to imagine a proper recursion mechanism. It seems difficult to define a general primitive recursive operator. Thus we have two recursive operators *Case* and *It*.
- Secondly, after defining recursive operators such as *Case* and *It*, we have to take care not to lose the property of adequacy of the encoding.

* Joelle.Despeyroux@sophia.inria.fr, LeleuP@wanadoo.fr. INRIA, 2004 Route des Lucioles, B.P. 93, F-06902 Sophia-Antipolis Cedex, France.

Let us elaborate a little bit on the problem of adequacy of representations. A priori, we can write *exotic terms*, i.e. terms that do not represent any term at the object level, such as the following term which describes an identity function using case analysis:

$$(\text{lam } \lambda x : \text{term}.(\text{Case } x \text{ } of \text{ } (\text{app } a \text{ } b) \Rightarrow (\text{app } a \text{ } b) \mid (\text{lam } e) \Rightarrow (\text{lam } e)))$$

This term is in normal form, and yet contains the Case operator which does not represent any construct in the object language -here the untyped λ-terms.

The modal connector $\Box$ allows us to distinguish the type of the general functions (arbitrary functional terms including those built with Case and It) from the type of the *parametric* functions (in the simple cases trees built over the meta level λ operator and the object level contructors of a language). We attribute types of the form $\Box A \rightarrow B$ to general functions whereas parametric functions keep their types $A \rightarrow B$. Thus, the above term involving a case analysis must now have the shape $\lambda x : \Box\text{term}.(\text{Case } x \text{ } of \text{ } \cdots)$ which has type $\Box term \rightarrow term$ and can no longer be accepted has a son of the *lam* constructor (which still has type $term \rightarrow term$). Since we can prove that the resulting system is a conservative extension of the simply-typed λ-calculus, the property of adequacy of the encoding is ensured.

Intuitively, we interpret $\Box A$ as the type of *closed* objects of type A. Structural recursion can proceed in a well-founding fashion on such objects, since all constructors are statically known. This is not the case if we may find an unknown free variable in the object.

Following this method, we have presented in [Lel97,DL98] a variant of [DPS97] with a modal core more comfortable to use ([Lel98,DL99]), defined by more compact typing rules, with reduction rules instead of evaluation rules. We have combined and adapted classical proof techniques to prove its metatheoretical properties. What we present in this paper is an extension of this system to dependent types.

First, we informally present the system by introducing some examples. Then, we present more rigorously its syntax, its typing rules and its reduction rules. Finally we prove some of its properties.

2 Motivations

2.1 Higher Order Abstract Syntax

The principle of HOAS is to represent object variables by meta variables, i.e. variables in the Logical Framework used to encode the logic, or language (called the object logic/language) under consideration. For instance, an untyped λ-term such as $\lambda y.(x \ y)$ is represented at the meta level by $\lambda x : term.(lam \lambda y : term.(app \ x \ y))$, where a meta variable x represents the object variable x and λ is the meta-level abstraction.

In [DH94], we introduced what we called the *higher-order terms*, which are functions from an arbitrary number of variables to terms. In [DFH95], we also considered functions from a *fixed* number of variables (when providing a meta-theoretical account of these terms). To recall the definition of those higher-order terms, let us consider as an example the following untyped λterms built on a predefined set of variables:

$$\begin{array}{l} \text{var}_v : \text{var} \rightarrow \text{term}_v \\ \text{app}_v : \text{term}_v \rightarrow \text{term}_v \rightarrow \text{term}_v \\ \text{lam}_v : (\text{var} \rightarrow \text{term}_v) \rightarrow \text{term}_v \end{array}$$

The higher-order terms corresponding to terms in $term_v$, belong to the type:

$$term_{v,n} = var \rightarrow \cdots \rightarrow var \rightarrow term.$$

The higher-order terms are defined with the help of what we called *higher-order constructors*. For the same example, these (pseudo) constructors are defined as follows:

Definition $\mathcal{R}ef_v = \lambda n : nat.\lambda i \in [0..n-1].\lambda x_{n-1}, \ldots, x_0 : var.(var_v \ x_i)$
Definition $\mathcal{A}pp_v = \lambda n : nat.\lambda e, e' : term_{v,n}.\lambda x_{n-1}, \ldots, x_0 : var. \ (app_v \ (e \ x_{n-1} \ \ldots \ x_0) \ (e' \ x_{n-1} \ \ldots \ x_0))$
Definition $\mathcal{L}am_v = \lambda n : nat.\lambda e : term_{v,n}.\lambda x_{n-1}, \ldots, x_0 : var. \ (lam_v \ \lambda y : var. \ (e \ y \ x_{n-1} \ \ldots \ x_0))$.

Our later meta-theoretical studies of various systems [Lel97,DL98], the lattest of which being presenting in the present paper, make extensive use of these higher-order terms. More precisely we used there (and use here) the following variant of $term_{v,n}$:

$$term_n = term \to \cdots \to term \to term.$$

Higher-order constructors for $term_n$ are defined similarly to higher-order constructors for $term_{v,n}$:

Definition $\mathcal{R}ef = \lambda n : nat.\lambda i \in [0..n-1].\lambda x_{n-1}, \ldots, x_0 : term.(var\ x_i)$
Definition $\mathcal{A}pp = \lambda n : nat.\lambda e, e' : term_n.\lambda x_{n-1}, \ldots, x_0 : term.\ (app\ (e\ x_{n-1}\ \ldots\ x_0)\ (e'\ x_{n-1}\ \ldots\ x_0))$
Definition $\mathcal{L}am = \lambda n : nat.\lambda e : term_n.\lambda x_{n-1}, \ldots, x_0 : term.\ (lam\ \lambda y : term.\ (e\ y\ x_{n-1}\ \ldots\ x_0)).$

Thus, in our view of HOAS [DH94,DFH95], an untyped λ-term with n free variables in term is represented at the meta level by a term of type $term_n$. We use vectorial notations $\overrightarrow{x} : \overrightarrow{term}$ for $x_1 : term, \ldots, x_n : term$.

Applications will be represented by terms of the form:

$$\lambda \overrightarrow{x} : \overrightarrow{\text{term}}.(\text{app}\ (M\ \overrightarrow{x})\ (N\ \overrightarrow{x}))$$

Abstractions will be represented by terms of the form:

$$\lambda \overrightarrow{x} : \overrightarrow{\text{term}}.(\text{lam}\ (M\ \overrightarrow{x}))$$

Free variables will be represented by projections:

$$\lambda \overrightarrow{x} : \overrightarrow{\text{term}}.x_i$$

2.2 An example using dependent types: The simply-typed λ-calculus

Thus we have seen how we can represent the type of the untyped λ-terms in the simply-typed λ-calculus. When we add dependent types in the Logical Framework, we are able to represent simply-typed λ-terms quite similarly:

```
(* Types *)
Inductive tp : Set :=
  b : tp
| arrow : tp -> tp -> tp.

(* Terms *)
HoInductive term : tp -> Set :=
  app : (A,B : tp) (term (arrow A B)) -> (term A) -> (term B)
| lam : (A,B : tp) ((term A) -> (term B)) -> (term (arrow A B)).
```

To define these types, we have used Coq syntax and a new imaginary keyword: *HoInductive*, which would be used to define inductive types using HOAS.

With dependent types, we can also define predicates like, for instance, predicate is_abs:

```
HoInductive is_abs : (A:tp) (term A) -> Prop :=
   lam_is_abs : (A,B:tp) (M:(term A)->(term B)) (is_abs (arrow A B) (lam A B M)).
```

In our view of HOAS, a simply-typed λ-term such as $\lambda y : A.(x\ y)$ where x is a variable of type $A \to B$ is represented at the meta level by $\lambda x : (term\ (arrow\ A\ B)).(lam\ A\ B\ \lambda y : (term\ A).(app\ A\ B\ x\ y))$.

More generally, a simply-typed λ-term of type B, with n free variables of types $A_1, \ldots, A_n$ will be represented at the meta level by a term of type

$$(\text{term}\ A_1) \to \cdots \to (\text{term}\ A_n) \to (\text{term}\ B).$$

We use vectorial notations $\overrightarrow{x} : \overrightarrow{A}$ for $x_1 : A_1, \ldots, x_n : A_n$.

Applications will be represented by terms of the form:

$$\lambda \overrightarrow{x} : \overrightarrow{(\text{term } A)}.(\text{app } \overrightarrow{A} \ B \ (M \ \overrightarrow{x}) \ (N \ \overrightarrow{x}))$$

Abstractions will be represented by terms of the form:

$$\lambda \overrightarrow{x} : \overrightarrow{(\text{term } A)}.(\text{lam } \overrightarrow{A} \ B \ (M \ \overrightarrow{x}))$$

Free variables will be represented by projections:

$$\lambda \overrightarrow{x} : \overrightarrow{(\text{term } A)}.x_i$$

2.3 Recursion

As we shall discuss in the conclusion, it seems that providing a primitive recursion construct here would provide us with an (internalized) induction principle for terms defined by means of HOAS; which, to our knowledge, nobody has been able to provide (and prove correct) up to now.

Thus in our previous works, we defined operators for iteration and case distinctions. Together with pairs, this gives tools as close as possible to primitive recursion.

In this section, we illustrate the behaviour of our operators for iteration and case reasoning by means of simple examples.

Example 1. Addition Among the possible types for addition, the most interesting are $\Box nat \to nat \to nat$ and $\Box nat \to \Box nat \to \Box nat$.

Informally, addition can be described by the following rules:

$$\begin{array}{l} \text{plus } 0 \ n = n \\ \text{plus } (s \ m) \ n = s \ (\text{plus } m \ n) \end{array}$$

In our system, as in previous ones [DPS97,DL98], we view iteration as replacing constructors of inductive types by functions of appropriate type. In the case of natural numbers, we replace $0 : nat$ by a term $M_0 : A$ and $s : nat \to nat$ by a function $M_s : A \to A$, when performing iteration with results in A. Thus iteration in this case replaces type nat by A. Iteration is written as

$$It \ (I, A, M) \ of \ \overrightarrow{f_i}$$

where M of type I is the subject of iteration and each f_i defines the term replacement for the corresponding constructor i of inductive type I; A being in the simple cases the result type of the iteration. In the rest of this section, we shall omit both types I and A, when these will be clear from the context.

Addition of type $\Box nat \to \Box nat \to \Box nat$ can be described in our system as follows:

$$plus = \lambda m : \Box nat.\lambda n : \Box nat.(It \ m \ of \ M_0 \ M_s)$$

where

$$\begin{array}{l} M_0 = n \\ M_s = \lambda x.\Box nat. \uparrow (s \ \downarrow x) \end{array}$$

and $\uparrow$ and $\downarrow$ are respectively the constructors and the destructors for $\Box$.

Example 2. Function *nbgvar* Function *nbgvar* counts the number of occurrences of variables of ground types in a simply-typed λ-term.

Informally, this function is defined by the following set of equations, for b a base type:

$$\begin{array}{l}\mathrm{nbgvar}(x) = \text{ if } x \text{ is of type } b \text{ then } 1 \text{ else } 0\\ \mathrm{nbgvar}(m\ n) = \mathrm{nbgvar}(m) + \mathrm{nbgvar}(n)\\ \mathrm{nbgvar}(\lambda x : a.(m\ x)) = \mathrm{nbgvar}(m\ x) \text{ with } \mathrm{nbgvar}(x) = \text{ if } x \text{ is of type } b \text{ then } 1 \text{ else } 0\end{array}$$

This informal definition shows that the function can formally be defined by iteration:

$$\mathrm{nbgvar} := \lambda a : \mathrm{tp}.\lambda m : \Box(\mathrm{term}\ a).(\mathrm{It}\ m \text{ of } \mathrm{M}'_{\mathrm{app}}\mathrm{M}'_{\mathrm{lam}}) : (a : \mathrm{tp})\ \Box(\mathrm{term}\ a) \rightarrow \Box\mathrm{nat}$$

where

$$\begin{array}{l}\mathrm{M}'_{\mathrm{app}} = \lambda a, b : \mathrm{tp}.\lambda p, q : \Box\mathrm{nat}.(\mathrm{plus}\ p\ q)\\ \mathrm{M}'_{\mathrm{lam}} = \lambda a, b : \mathrm{tp}.\lambda f : \Box\mathrm{nat} \rightarrow \Box\mathrm{nat}.(f\ (\mathrm{Ground}\ a))\end{array}$$

where *Ground* is the function from types to integers such that $\mathrm{Ground}(b) =\uparrow 1$ for b a ground type and otherwise $\mathrm{Ground}(t) =\uparrow 0$ for $t : \mathrm{tp}$ not a ground type.

Example 3. Function *form* The function *form* giving the form of a simply-typed λ-term is informally defined as follows:

$$\begin{array}{l}\mathrm{form}(\lambda x.x) = 0\\ \mathrm{form}(\lambda x.(m\ x)\ (n\ x)) = 1\\ \mathrm{form}(\lambda x.\lambda y : a.(m\ x\ y)) = 2\end{array}$$

Note that, as in previous work [DPS97,DL98], we had to *close* the argument to be able to perform case analysis on it. In the formal setting, this means that we have to define a function *form* of type $(a : tp)\Box((term\ a) \rightarrow (term\ a)) \rightarrow nat$, which will consist on a function (the case analysis) applied to 0 - the value for variables:

$$\begin{array}{rl}form = & \lambda a : tp.\lambda m : \Box((term\ a) \rightarrow (term\ a)).(Case\ m\ M_{app}\ M_{lam}\ \ 0)\\ : & (a : tp)\Box((term\ a) \rightarrow (term\ a)) \rightarrow nat\end{array}$$

where

$$\begin{array}{l}M_{app} = \lambda a', b' : tp.\lambda x : \Box((term\ a) \rightarrow (term\ (arrow\ a'\ b'))).\lambda y : \Box((term\ a) \rightarrow (term\ a')).1\\ M_{lam} = \lambda a', b' : tp.\lambda f : \Box((term\ a) \rightarrow (term\ a') \rightarrow (term\ b')).2\end{array}$$

2.4 Recursion on Terms of Dependent Types

The main difficulty lying ahead is to find a recursion mechanism for types defined by HOAS. As for adequacy of the encoding, we can hope that the modal core will still ensure it, like in [DL98].

To illustrate the problems introduced by dependent types, let us consider the following term:

$$M := \lambda\sigma : \mathrm{tp}.\lambda x : (\mathrm{term}\ \sigma).\lambda\tau : \mathrm{tp}.(\mathrm{lam}\ \tau\ \sigma\ \lambda z : (\mathrm{term}\ \tau).x)$$

Variables σ, τ and x are not of the same kind: the former ones appear in types, not the latter one. During *Case/It* reduction, we need to keep σ and τ, which is not enabled by the rules of [DL98]. Thus a naive extension of the rules of [DL98] does not work.

To solve this problem, we distinguish variables such as σ and τ from variable such as x. The former ones, called *dependent* variables, appear in types. On the contrary, the variable x is a *pseudo constructor*. Its purpose is made clear by the representation mechanism: it represents a free variable. *Dependent* *variables* deserve a special treatment so that we can reuse them in types. The other variables are processed like in [DL98]. In our example, the reduction rules for $Case$ and It will be:

$$
\begin{array}{l}
(\mathrm{Case}\ \uparrow \lambda\sigma : \mathrm{tp}.\lambda x : (\mathrm{term}\ \sigma).\lambda\tau : \mathrm{tp}.(\mathrm{lam}\ \tau\ \sigma\ \lambda z : (\mathrm{term}\ \tau).x)) \hookrightarrow \\
\lambda\sigma' : \mathrm{tp}.\lambda x' : (Q\ \sigma').\lambda\tau' : \mathrm{tp}. \\
(\mathrm{M}_{\mathrm{lam}}\ \tau' \\
\qquad \sigma' \\
\qquad \uparrow \lambda\sigma : \mathrm{tp}.\lambda x : (\mathrm{term}\ \sigma).\lambda\tau : \mathrm{tp}.\lambda z : (\mathrm{term}\ \tau).x)
\end{array}
$$

$$
\begin{array}{l}
(\mathrm{It}\ \uparrow \lambda\sigma : \mathrm{tp}.\lambda x : (\mathrm{term}\ \sigma).\lambda\tau : \mathrm{tp}.(\mathrm{lam}\ \tau\ \sigma\ \lambda z : (\mathrm{term}\ \tau).x)) \hookrightarrow \\
\lambda\sigma' : \mathrm{tp}.\lambda x' : (Q\ \sigma).\lambda\tau' : \mathrm{tp}. \\
(\mathrm{M}'_{\mathrm{lam}}\ \tau' \\
\qquad \sigma' \\
\qquad (\mathrm{It}\ \uparrow \lambda\sigma : \mathrm{tp}.\lambda x : (\mathrm{term}\ \sigma).\lambda\tau : \mathrm{tp}.\lambda z : (\mathrm{term}\ \tau).x\ \ \sigma'\ x'\ \tau'))
\end{array}
$$

The types of the dependent variables are preserved by these reduction rules.

3 The System

Now we turn to the presentation of our system. In the following, our notations are close to those used in the usual presentations of the Calculus of Inductive Constructions (CIC). In [DL98], we have defined inductive types and their constructors from an arbitrary signature, which enabled mutually defined inductive types. Here, for the sake of simplicity, we forbid this possibility and we define the inductive types one after another, like in the first presentations of CIC ([Wer94],[PM92]). Our typing rules are rather similar to those of CIC and the similarity of the notations makes the comparison easier.

3.1 Syntax

Terms Our system has two *sorts*:

$$s := \mathrm{Set} \mid \mathrm{Type}$$

Like in usual presentations of the Pure Typed Systems [Bar91], we do not distinguish terms from types :

$$
\begin{array}{rl}
t := & s \mid x \mid (x:t)t \mid \lambda x : t.t \mid (t\ t) \mid \Box t \mid\ \uparrow t \mid\ \downarrow t \\
 & \mid \mathrm{Ind}(x:T)\{\overrightarrow{t}\} \mid \mathrm{Constr}(i,t) \mid \mathrm{Case}\ t\ \mathrm{of}\ \overrightarrow{t} \mid \mathrm{It}\ t\ \mathrm{of}\ \overrightarrow{t}
\end{array}
$$

where x is a variable, i is a positive integer and $\overrightarrow{t}$ is defined by : $\overrightarrow{t} := \{\} \mid t :: \overrightarrow{t}$ (vectorial notations).

The notation $m \to n$ is an abbreviation for the product $(x : m)n$ whenever x does not occur in n. As usual the product associates to the right while the application associates to the left.

Examples All examples of (non-mutual) inductive definitions in CIC can be written here. As a typical example, let us recall the definition of lists of elements of a predefined type A, with a given length. It is an inductive type $list$ of type $nat \to Set$ defined as:

$$list \equiv \mathrm{Ind}(X : nat \to Set)\ (X\ 0) \mid (n : nat)\ A \to (X\ n) \to (X\ (s\ n)) :\ nat \to Set$$

with the following constructors:

$$
\begin{array}{l}
nil \equiv\ \ Constr(1, list) : (list\ 0) \\
cons \equiv Constr(2, list) : (n : nat)\ A \to (list\ n) \to (list\ (s\ n))
\end{array}
$$

In addition to these examples, we can define inductive types involving HOAS, as the untyped λ-terms given in the begining of the present paper:

$$term \equiv \mathrm{Ind}(X : Set)\ X \to X \to X \mid (X \to X) \to X :\ Set$$

with the following constructors:

$$\begin{array}{l} app \equiv\ Constr(1, term) : term \to term \to term \\ lam \equiv Constr(2, term) : (term \to term) \to term \end{array}$$

Technically, those examples are not allowed in CIC, because of the negative (first) occurrence of *term* in the type of *lam*, as we pointed out in the introduction. We succeeded in representing them in our system (as in the previous versions of it without dependent types), thanks to the separation of the function space $A \to B$, used to represent syntaxes, from the function space $\Box A \to B$, used to define recursion on inductive types.

Context stacks There are many different presentations of the modal λ-calculus in the literature. We have chosen here Frank Pfenning and Hao-Chi Wong's presentation ([PW95]). It involves context stacks $\Delta := \Gamma_1; \ldots; \Gamma_n$ instead of usual contexts Γ. If we adopt a temporal interpretation of modal logic IS4, we can see each context as a moment in time, Γ_n containing the present hypotheses. The main advantage of this system is its simplicity. We do not have *let* construct like in [DPS97]. Instead, we just have a term constructor $\uparrow$ and a term destructor $\downarrow$ for $\Box$. In the metatheoretical proofs, we manage to overcome the fact that the typing rules of the system are not syntax driven and that the judgments involve context stacks (for the metatheoretical properties of modal λ-calculus IS4, cf. [Lel98,DL99]).

3.2 Definitions

The definition for the forms of the constructors is the same as the one used in CIC except that we forget the positivity constraint ([PM92]) when checking that a constructor has a valid type, in order to allow terms defined in HOAS. Notice that a constructor does not contain any modal construction.

Definition 1 (Forms of constructors). The *forms of constructors* with respect to X are generated by the following syntax, where m and M are terms:

$$C := X \mid (C\ m) \mid C \to C \mid (x : M)C$$

with the constraint that x does not occur in M and m, like in CIC. We say that C is a *type of constructor* in the case C is a type (i.e its type is a sort).

When we iteratively traverse a term of inductive type I which contains a higher order constructor C, we have to consider functional terms such as $\lambda \overrightarrow{z_i} : \overrightarrow{B_i}.M$ where the types B_i have a certain form inherited from the type of C.

For instance, with the notations introduced in section 2.2, the third son of the constructor *lam* in the term (lam $b\ b\ \lambda x$: (term b).x) is the functional term λx : (term b).x.

If the type B_i contains the inductive type I, the variable z_i is called a pseudo constructor (since it can play the same role as a constructor). The following definition, quite similar to the previous one, characterizes its type:

Definition 2 (Forms of pseudo constructors). The terms B which are *forms of pseudo constructors* for inductive type I and signature $\{z_i : B_i\}$ are:

- the terms generated by the following syntax:

  $$B := I \mid (B\ m) \mid B \to B \mid (x : M)B$$

  with the constraints that I appears neither in M nor in m, and that neither M nor m contains a z_i whose type contains I.

– and the terms which do not contain I.

Moreover, B is called a *type of pseudo constructor* if the type of B is a sort.

The forms of pseudo constructor are defined such that when we recursively traverse a closed term of inductive type, we get terms $\lambda \overrightarrow{z_i} : \overrightarrow{B_i}.M$, where each type B_i is a form of pseudo constructor for I and for $\{z_j : B_j\}$ (where $j < i$).

For instance, if we compute a result by iteration over the argument (lam b b λx : (term b).x), after an iteration step, we have to process the argument λx : (term b).x, where x is a pseudo constructor of type (term b), which is indeed a type of pseudo constructor for type term and the empty signature.

The constraint on the variables z_i in the definition of the forms of pseudo constructors will turn out to be essential when verifying the property of preservation of typing by the evaluation.

Definition 3 (Dependent arguments of a constructor).

Given the constructor C of type T and the term $(C\ \overrightarrow{M})$, where $(C\ \overrightarrow{M})$ is saturated (i.e. we have provided all the arguments of C), we have given all the arguments of M), $\mathrm{dep}(\overrightarrow{M}, T)$ is the list of terms inductively defined as follows:

– $\mathrm{dep}(\overrightarrow{M}, (X\ \overrightarrow{t})) = ()$
– $\mathrm{dep}(\overrightarrow{M}, (x : t)D) = \begin{cases} M_1 \cup \mathrm{dep}(\overrightarrow{M_2 \ldots M_n}, D) \text{ if } x \text{ appears in } D, \\ \mathrm{dep}(\overrightarrow{M_2 \ldots M_n}, D) \text{ otherwise} \end{cases}$
– $\mathrm{dep}(\overrightarrow{M}, D_1 \to D_2) = \mathrm{dep}(\overrightarrow{M_2 \ldots M_n}, D_2)$

Example In the following, A and B are the dependent arguments of constructors *app* and *lam*:

```
app : (A,B : tp) (term (arrow A B)) -> (term A) -> (term B)
lam : (A,B : tp) ((term A) -> (term B)) -> (term (arrow A B))
```

3.3 Typing Rules

The typing rules are those of CC with extra modal rules and rules for the *Case* and *It* operators.

Rules of CC s denotates Set or Type.

$$(\mathrm{Ax})\ \frac{}{.\vdash \mathrm{Set} : \mathrm{Type}}$$

$$(\mathrm{Prod})\ \frac{\Delta, x : t_1 \vdash t_2 : s}{\Delta \vdash (x : t_1)t_2 : s} \qquad (\mathrm{Lam})\ \frac{\Delta \vdash (x : t_1)t_2 : s \qquad \Delta, x : t_1 \vdash t : t_2}{\Delta \vdash \lambda x : t_1.t : (x : t_1)t_2}$$

$$(\mathrm{Aff})\ \frac{\Delta \vdash t : s \qquad \Delta \vdash A : B \qquad x \notin \Delta}{\Delta, x : t \vdash A : B}$$

$$(\mathrm{Var})\ \frac{\Delta \vdash t : s \qquad (x : t) \in \Delta}{\Delta \vdash x : t} \qquad (\mathrm{App})\ \frac{\Delta \vdash t_2 : (x : T_1)T_2 \qquad \Delta \vdash t_1 : T_1}{\Delta \vdash (t_2\ t_1) : T_2[t_1/x]}$$

$$(\mathrm{Conv})\ \frac{\Delta \vdash t : T_1 \qquad \Delta \vdash T_1 : s \qquad \Delta \vdash T_2 : s \qquad T_1 =_{\beta\eta} T_2}{\Delta \vdash t : T_2}$$

Modal rules

The modal rules are those of [PW95].

$$\text{(Pop)}\ \frac{\Delta \vdash M : \Box A}{\Delta ; \Gamma \vdash M : \Box A} \qquad (\uparrow)\ \frac{\Delta ; . \vdash M : A}{\Delta \vdash \uparrow M : \Box A} \qquad (\downarrow)\ \frac{\Delta \vdash M : \Box A}{\Delta \vdash \downarrow M : A}$$

Case Operator

$$\text{(Case)}\ \frac{\begin{array}{ll} B_i \text{ forms of pseudo constructor} & \\ \text{for } I \text{ and } \{z_j : B_j\}_{j<i} & \\ A \equiv (\overrightarrow{x} : \overrightarrow{A})\text{Set} & I \equiv \text{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \\ \Delta \vdash \overrightarrow{u} : (\overrightarrow{x} : \overrightarrow{A}) & \Delta \vdash t : \Box(\overrightarrow{z_i} : \overrightarrow{B_i})(I\ \overrightarrow{u}) \\ \Delta \vdash Q : (\overrightarrow{x} : \overrightarrow{A})\text{Set} & \forall i \in \mathbb{N}\ , \Delta \vdash f_i : \Delta_C\{C_i(I), Q\} \end{array}}{\Delta \vdash \text{Case}(I, Q, t)\{\overrightarrow{f_i}\} : (\overrightarrow{z_i} : \overrightarrow{H_C(B_i, Q)})(Q\ \overrightarrow{u})}$$

where the type $\Delta_C\{C_i(I), Q\}$ is defined as follows :

- $\Delta_C\{(X\ t), Q\} = (Q\ t)$
- $\Delta_C\{(x : t)D, Q\} = (x : t)\Delta_C\{D, Q\}$
- $\Delta_C\{D_1 \to D_2, Q\} = \Box\left[(\overrightarrow{z_i} : \overrightarrow{B_i})D_1\right] \to \Delta_C\{D_2, Q\}$

$H_C(B_i, Q)$ is simply equal to B_i if B_i does not contain I and otherwise $H_C(B_i, Q)$ is defined like Δ_C:

- $H_C((I\ m), Q) = (Q\ m)$
- $H_C((x : t)D, Q) = (x : t)H_C(D, Q)$
- $H_C(D_1 \to D_2, Q) = \Box\left[(\overrightarrow{z_i} : \overrightarrow{B_i})D_1\right] \to H_C\{D_2, Q\}$

It Operator

$$\text{(It)}\ \frac{\begin{array}{ll} B_i \text{ forms of pseudo constructor} & \\ \text{for } I \text{ and } \{z_j : B_j\}_{j<i} & \\ A \equiv (\overrightarrow{x} : \overrightarrow{A})\text{Set} & I \equiv \text{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \\ \Delta \vdash \overrightarrow{u} : (\overrightarrow{x} : \overrightarrow{A}) & \Delta \vdash t : \Box(\overrightarrow{z_i} : \overrightarrow{B_i})(I\ \overrightarrow{u}) \\ \Delta \vdash Q : (\overrightarrow{x} : \overrightarrow{A})\text{Set} & \forall i \in \mathbb{N}\ , \Delta \vdash f_i : \Delta_I\{C_i(I), Q\} \end{array}}{\Delta \vdash \text{It}(I, Q, t)\{\overrightarrow{f_i}\} : (\overrightarrow{z_i} : \overrightarrow{H_I(B_i, Q)})(Q\ \overrightarrow{u})}$$

where the type $\Delta_I\{C_i(I), Q\}$ is defined as follows:

- $\Delta_I\{(X\ t), Q\} = (Q\ t)$
- $\Delta_I\{(x : t)D, Q\} = (x : t)\Delta_I\{D, Q\}$
- $\Delta_I\{D_1 \to D_2, Q\} = \Delta_I\{D_1, Q\} \to \Delta_I\{D_2, Q\}$

$H_I(B_i, Q)$ is simply equal to B_i if B_i does not contain I and otherwise $H_I(B_i, Q)$ is defined like Δ_I:

- $H_I((I\ m), Q) = (Q\ m)$
- $H_I((x : t)D, Q) = (x : t)H_I(D, Q)$
- $H_I(D_1 \to D_2, Q) = H_I(D_1, Q) \to H_I(D_2, Q)$

In other words, Δ_I and H_I substitute Q for the occurrences of X or I.

Examples

The typing rules for $Case$ and It are illustrated by the examples of section 2.3.

For case analysis, let us recall the $form$ example:

$$\begin{array}{rl} form = & \lambda a : tp.\lambda m : \Box((term\ a) \to (term\ a)).(Case\ m\ M_{app}\ M_{lam}\ \ 0) \\ : & (a : tp)\Box((term\ a) \to (term\ a)) \to nat \end{array}$$

where

$$\begin{array}{l} \mathrm{M_{app}} = \lambda a', b' : tp.\lambda x : \Box((term\ a) \to (term\ (arrow\ a'\ b'))).\lambda y : \Box((term\ a) \to (term\ a')).1 \\ \mathrm{M_{lam}} = \lambda a', b' : tp.\lambda f : \Box((term\ a) \to (term\ a') \to (term\ b')).2 \end{array}$$

We have

$$\begin{array}{l} t : \Box((term\ a) \to (term\ a)) \\ Q : \lambda a : tp.nat \\ C_{app}(term) = Constr(1, term) = (a, b : tp)(term\ (arrow\ a\ b)) \to (term\ a) \to (term\ b) \\ C_{lam}(term) = Constr(2, term) = (a, b : tp)((term\ a) \to (term\ b)) \to (term\ (arrow\ a\ b)) \\ \mathrm{M_{app}} : \Delta_C\{C_{app}(term), Q\} = \\ \qquad (a', b' : tp)\Box((term\ a) \to (term\ (arrow\ a'\ b'))) \to \Box((term\ a) \to (term\ a')) \to nat \\ \mathrm{M_{lam}} : \Delta_C\{C_{lam}(term), Q\} = (a', b' : tp)\Box((term\ a) \to (term\ a') \to (term\ b')) \to nat \\ Case\ t\ \mathrm{M_{app}}\ \mathrm{M_{lam}} :\ (H_C((term\ a), Q) \to (Q\ a) = (Q\ a) \to (Q\ a) = nat \to nat \end{array}$$

For iteration, let us consider the $nbgvar$ example, which counts the number of occurrences of variables of ground types in a simply-typed λ-term:

$$\mathrm{nbgvar} := \lambda a : \mathrm{tp}.\lambda t : \Box(\mathrm{term}\ a).(\mathrm{It}\ t\ \mathrm{of}\ \mathrm{M'_{app}M'_{lam}}) : (a : \mathrm{tp})\ \Box(\mathrm{term}\ a) \to \Box\mathrm{nat}$$

where

$$\begin{array}{l} \mathrm{M'_{app}} = \lambda a, b : \mathrm{tp}.\lambda p, q : \Box\mathrm{nat}.(\mathrm{plus}\ p\ q) \\ \mathrm{M'_{lam}} = \lambda a, b : \mathrm{tp}.\lambda f : \Box\mathrm{nat} \to \Box\mathrm{nat}.(f\ (\mathrm{Ground}\ a)) \end{array}$$

We have

$$\begin{array}{l} t : \Box(\mathrm{term}\ a) \\ Q : \lambda a : tp.\Box nat \\ C_{app}(term) = Constr(1, term) = (a, b : tp)(term\ (arrow\ a\ b)) \to (term\ a) \to (term\ b) \\ C_{lam}(term) = Constr(2, term) = (a, b : tp)((term\ a) \to (term\ b)) \to (term\ (arrow\ a\ b)) \\ \mathrm{M'_{app}} : \Delta_I\{C_{app}(term), Q\} = (a, b : tp)\Box\mathrm{nat} \to \Box\mathrm{nat} \to \Box\mathrm{nat} \\ \mathrm{M'_{lam}} : \Delta_I\{C_{lam}(term), Q\} = (a, b : tp)(\Box\mathrm{nat} \to \Box\mathrm{nat}) \to \Box\mathrm{nat} \\ \mathrm{It}\ t\ \mathrm{of}\ \mathrm{M'_{app}M'_{lam}} :\ (Q\ a) = \Box nat \end{array}$$

3.4 Reduction Rules

In this section, we give the reduction rules of the system. They include a rule for η-expansion (instead of η-contraction) because what we are really interested in are the normal η-long forms, which are in bijection with the entities we represent. Fortunately, η-expansion behaves nicely with rewriting rules such as the reduction rules for $Case$ and It.

β and η rules

$$(\beta\ )\ \frac{\Delta, x : A \vdash M : B \qquad \Delta \vdash N : A}{\Delta \vdash (\lambda x : A.M\ N) \hookrightarrow M[N/x] : B}$$

$$(\eta)\ \frac{\Delta \vdash M : A \to B}{\Delta \vdash M \hookrightarrow \lambda x : A.(M\ x) : A \to B}\ \text{if } x \text{ is not free in } M$$

Modal rules

$$(\beta\Box)\ \frac{\Delta \vdash \downarrow\uparrow M : A}{\Delta \vdash \downarrow\uparrow M \hookrightarrow M : A} \qquad (\eta\Box)\ \frac{\Delta \vdash \uparrow\downarrow M : \Box A}{\Delta \vdash \uparrow\downarrow M \hookrightarrow M : \Box A}$$

Case operator For the sake of readability, we present here the untyped versions of the reduction rules for the *Case*/*It* operators. The typed versions can easily be obtained from the typing rules of section 3.5.

To ensure preservation of typing, we must add a constraint: the variables z_i which appear in a dependent argument of the constructor cannot have a type which contains I (because the variables z_i do not have type B_i on the right side of the reduction rule but rather $H_C(B_i, Q)$).

$$\begin{array}{l} \mathrm{Case}(I, Q, \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.(\mathrm{Constr}(I,k)\ M_1 \ldots M_n))\{\overrightarrow{f_i}\} \hookrightarrow \\ \lambda\overrightarrow{z_i} : \overrightarrow{H_C(B_i,Q)}. \left(f_k \left\{ \begin{array}{ll} \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.M_j & \text{if } M_j \text{ has target } I \\ M_j & \text{if } M_j \text{ does not have target } I \end{array} \right\}_{j=1\ldots n} \right) \end{array}$$

$$\begin{array}{l} \mathrm{Case}(I, Q, \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.(z_k\ M_1 \ldots M_n))\{\overrightarrow{f_i}\} \hookrightarrow \\ \lambda\overrightarrow{z_i} : \overrightarrow{H_C(B_i,Q)}. \left(z_k \left\{ \begin{array}{ll} \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.M_j & \text{if } M_j \text{ has target } I \\ M_j & \text{if } M_j \text{ does not have target } I \end{array} \right\}_{j=1\ldots n} \right) \end{array}$$

It operator Like in the case of the *Case* operator, we must add a constraint to have preservation of typing: the variables z_i which appear in a dependent argument of the constructor cannot have a type which contains I (because the variables z_i do not have type B_i on the right side of the reduction rule but rather $H_I(B_i, Q)$).

$$\begin{array}{l} \mathrm{It}(I, Q, \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.(\mathrm{Constr}(I,k)\ M_1 \ldots M_n))\{\overrightarrow{f_i}\} \hookrightarrow \\ \lambda\overrightarrow{z_i} : \overrightarrow{H_I(B_i,Q)}. \left(f_k \left\{ \begin{array}{ll} \mathrm{It}(I, Q, \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.M_j)\{\overrightarrow{f_i}\}\ z_i & \text{if } M_j \text{ has target } I \\ M_j & \text{if } M_j \text{ does not have target } I \end{array} \right\}_{j=1\ldots n} \right) \end{array}$$

$$\begin{array}{l} \mathrm{It}(I, Q, \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.(z_k\ M_1 \ldots M_n))\{\overrightarrow{f_i}\} \hookrightarrow \\ \lambda\overrightarrow{z_i} : \overrightarrow{H_I(B_i,Q)}. \left(z_k \left\{ \begin{array}{ll} \mathrm{It}(I, Q, \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.M_j)\{\overrightarrow{f_i}\}\ z_i & \text{if } M_j \text{ has target } I \\ M_j & \text{if } M_j \text{ does not have target } I \end{array} \right\}_{j=1\ldots n} \right) \end{array}$$

Discussion The side conditions of the reduction rules are essential. For example, let us try to apply the *Case* operator to the following term:

$$\sigma : \Box\mathrm{tp}, f : \Box((\mathrm{term}\ \ (\downarrow \sigma)) \to \mathrm{tp}) \vdash \uparrow \lambda x : (\mathrm{term}\ (\downarrow \sigma)).(\mathrm{lam}\ ((\downarrow f)\ x)\ \ (\downarrow \sigma)\ \ \lambda z : (\mathrm{term}\ ((\downarrow f)\ x)).x)$$

After a reduction step, we would get:

$$\lambda x : (Q\ \ (\downarrow \sigma)).(\mathrm{M_{lam}}\ ((\downarrow f)\ x)\ \ (\downarrow \sigma)\ \ \uparrow \lambda x : (\mathrm{term}\ (\downarrow \sigma)).\lambda z : (\mathrm{term}\ ((\downarrow f)\ x)).x)$$

which is not well typed in the context stack $(\sigma : \Box\mathrm{tp}, f : \Box((\mathrm{term}\ \ (\downarrow \sigma)) \to \mathrm{tp}))$ because the type of x has changed during the reduction process.

3.5 Comparison with CIC

The contexts, terms and types of the system are those of the Calculus of Constructions (CC) and we have inductive mechanisms close to those of CIC, except that:

- because of the chosen presentation of $\lambda\Box$, we have context stacks instead of usual contexts,
- we have modal constructions: if T is a type then $\Box T$ is also a type. As for terms, we have a constructor and a destructor for $\Box$: $\uparrow$ and $\downarrow$,
- we have already noticed that we allow more inductive types: the constructors can contain negative occurrences of the inductive type they define,
- we do not have a primitive recursive operator as general as in CIC but instead we have two recursive operators *Case* and *It*. We only have a non dependent elimination rule.

Let us compare both systems in more details.

Inductive rules in CIC Our rules derive from the rules of [DL98] and from the rule of non dependent elimination (in the following Q belongs to a non-dependent type) of an inductive type in CIC, that we recall here:

$$(\text{Nodep})\ \frac{\begin{array}{ll} A \equiv (\overrightarrow{x} : \overrightarrow{A})\text{Set} & I \equiv \text{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \\ \Gamma \vdash \overrightarrow{u} : (\overrightarrow{x} : \overrightarrow{A}) & \Gamma \vdash t : (I\ \overrightarrow{u}) \\ \multicolumn{2}{l}{\Gamma \vdash Q : (\overrightarrow{x} : \overrightarrow{A})\text{Set}\ \forall i \in \mathbb{N}\ , \Gamma \vdash f_i : \Delta\{C_i(I), Q\}} \end{array}}{\Gamma \vdash \text{Elim}(t, Q)\{\overrightarrow{f_i}\} : (Q\ \overrightarrow{u})}$$

where $\Delta\{C_i(I), Q\}$ is defined by the following equalities:

$$\begin{aligned} \Delta\{(X\ \overrightarrow{t}), Q\} &\equiv (Q\ \overrightarrow{t}) \\ \Delta\{(x : t)D(X), Q\} &\equiv (x : t)\Delta\{D(X), Q\} \\ \Delta\{((\overrightarrow{x} : \overrightarrow{t})(X\ \overrightarrow{t'}) \to D(X), Q\} &\equiv ((\overrightarrow{x} : \overrightarrow{t})(X\ \overrightarrow{t'})) \to ((\overrightarrow{x} : \overrightarrow{t})(Q\ \overrightarrow{t'})) \to \Delta\{D(X), Q\} \end{aligned}$$

For instance, in CIC, for the inductive type $I = \text{nat}$, if $\Gamma \vdash Q : \text{Set}$, $\Gamma \vdash t : \text{nat}$, $\Gamma \vdash f_0 : A$ and $\Gamma \vdash f_S : \text{nat} \to A \to A$, then $\text{Elim}(t, Q)\{f_0, f_S\}$ has type A in Γ. Moreover, we have:

$$\begin{aligned} \text{Elim}(0, Q)\{f_0, f_S\} &= f_0 \\ \text{Elim}((S\ n), Q)\{f_0, f_S\} &= (f_S\ \ n\ \ \text{Elim}(n, Q)\{f_0, f_S\}) \end{aligned}$$

The *Elim* operator implements the mechanism of primitive recursion. In general, for an inductive type I which has a constructor C, $\text{Elim}((C\ M_1 \dots M_n), Q)\{f_i\}$ is computed from the terms $\text{Elim}(M_j, Q)\{f_i\}$ and M_j.

The operators *Case* and *It* of our calculus are a bit different. They do not implement full primitive recursion but case analysis and iteration. Thus, $\text{Case}((C\ M_1 \dots M_n), Q, t)\{\overrightarrow{f_i}\}$ will only be computed from the terms M_j. And $\text{It}((C\ M_1 \dots M_n), Q, t)\{\overrightarrow{f_i}\}$ will only be computed from the terms $\text{It}(M_j, Q, t)\{\overrightarrow{f_i}\}$.

Typing rules The main differences between the rules (Nodep) of CIC and our rule for case are:

- in rule (Case), the type of the main argument t involves modality and the collection of variables $(\overrightarrow{z_i} : \overrightarrow{B_i})$,
- the Case operator has a functional type,
- in rule (Case), $\Delta_C\{C_i(I), Q\}$ is built like $\Delta\{C_i(I), Q\}$ except that when calculating $\Delta_C\{D_1 \to D_2, Q\}$, we only introduce the type of the son of the constructor $\Box\left[(\overrightarrow{z_i} : \overrightarrow{B_i})D_1\right]$.

The differences between the rules (Nodep) of CIC and our rule for iteration are similar to the differences between the rules (Nodep) and (Case) except that when calculating $\Delta_I\{D_1 \to D_2, Q\}$, we only introduce the type of the value of the iteration for the son of the constructor $\Delta_I\{D_1, Q\}$.

Note The types involved in the rules (Case) and (It) are all well typed terms. A priori, this is not obvious for the type of the recursive operators: $(\overrightarrow{z_i} : \overrightarrow{H_C(B_i, Q)})(Q\ \overrightarrow{u})$ and $(\overrightarrow{z_i} : \overrightarrow{H_I(B_i, Q)})(Q\ \overrightarrow{u})$. This can be proved easily thanks to the side conditions B_i forms of pseudo constructor for I and $\{z_j : B_j\}_{j<i}$.

Reduction rules From the recursive operator of CIC, we can build a case analysis and an iteration operator (by forgetting some arguments). Contrary to the case analysis operator, the iteration operator of CIC behaves differently than ours. Let us see the differences on the ordinal numbers:

```
Inductive ord : Set := O_ord : ord | S_ord : ord -> ord | lim : (nat -> ord) -> ord.
```

Let us call $\mathrm{It}_{\mathrm{CIC}}$ the iteration operator in CIC. $\mathrm{It}_{\mathrm{CIC}}$ cannot deal with functions. Its main argument has to be a term of type ord. Thus, we have:

$$(\mathrm{It}_{\mathrm{CIC}}\ (\lim u)\ \mathrm{of}\ldots) = (M_{\lim}\ \lambda n : \mathrm{nat}.(\mathrm{It}_{\mathrm{CIC}}\ (u\ n)\ \mathrm{of}\ldots))$$

If we define the type *ord* in our system , we simply have :

$$(\mathrm{It}\ (\lim u)\ \mathrm{of}\ldots) = (M_{\lim}\ (\mathrm{It}\ u\ \mathrm{of}\ldots))$$

4 Properties

The proofs of the theorems presented here were not particularly difficult. The difficult task was to find rules for typing and evaluation making them feasible.

4.1 Preservation of Typing and Local Confluence

Theorem 4 (Preservation of typing). *If $\Delta \vdash M : A$ and $\Delta \vdash M \hookrightarrow N : A$ then $\Delta \vdash N : A$.*

Proof By induction on $\Delta \vdash M \hookrightarrow N : A$.

The cases of the rules (β), (η), $(\beta\Box)$ and $(\eta\Box)$ are easy. The subtle cases are (Case) et (It). These cases show the usefulness of the part of the side conditions dealing with the forms of the pseudo constructors in the rules. For instance,

$$\begin{array}{l}\mathrm{Case}(I, Q, \uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.(\mathrm{Constr}(I,k)\ M_1 \ldots M_n))\{\overrightarrow{f_i}\} \hookrightarrow \\ \lambda\overrightarrow{z_i} : \overrightarrow{H_C(B_i, Q)}. \left(f_k \left\{\begin{array}{ll}\uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.M_j & \text{if } M_j \text{ has target } I \\ M_j & \text{if } M_j \text{ does not have target } I\end{array}\right\}_{j=1\ldots n}\right)\end{array}$$

For each argument M_j of target I, f_k wants an argument of type $\Box\left[(\overrightarrow{z_i} : \overrightarrow{B_i})D\right]$. Now, we have:

$$\Delta \vdash\uparrow \lambda\overrightarrow{z_i} : \overrightarrow{B_i}.M_j : \Box\left[(\overrightarrow{z_i} : \overrightarrow{B_i})D\right]$$

For the other arguments M_k of type D (that do not have target I), f_k wants an argument of type D. We give it M_k. We know that:

$$\Delta, \overrightarrow{z_i} : \overrightarrow{B_i} \vdash M_k : D$$

We still have to check that:

$$\Delta, z_i : \overrightarrow{H_C(B_i, Q)} \vdash M_k : D$$

This judgment is true if the types B_i are invariant by H_C. This is true if the variables z_i which appear in M_k have no occurrence of type I in their types. This is exactly what the side conditions of the reduction rules state.

The case of the iteration operator is also interesting:

$$\begin{array}{l}\mathrm{It}(I,Q,\uparrow\lambda\overrightarrow{z_i}:\overrightarrow{B_i}.(\mathrm{Constr}(I,k)\ M_1\ldots M_n))\{\overrightarrow{f_i}\}\hookrightarrow\\ \lambda\overrightarrow{z_i}:\overrightarrow{H_I(B_i,Q)}.\left(f_k\ \left\{\begin{array}{ll}\mathrm{It}(I,Q,\uparrow\lambda\overrightarrow{z_i}:\overrightarrow{B_i}.M_j)\{\overrightarrow{f_i}\}\ z_i & \text{if } M_j \text{ has target } I\\ M_j & \text{if } M_j \text{ does not have target } I\end{array}\right\}_{j=1\ldots n}\right)\end{array}$$

The proof is similar to the case of the *Case* operator with an extra little difficulty: we have to show that the new $z'_j : B'_j$ introduced in M_j are such that B'_j is a form of pseudo constructor for I and $\{z_i : B_i, z'_k : B'_k\}(k < j)$. This is true because the dependent arguments are the only ones that can carry the variables z_i in the other arguments of the constructor. The side conditions of the reduction rules ensure that these dependent arguments do not contain any undesirable z_i, i.e. any z_i whose type would contain I.

□

Theorem 5 (Local confluence).
If $\Delta \vdash M \hookrightarrow N : A$ and $\Delta \vdash M \hookrightarrow P : A$ then there is a term Q such that $\Delta \vdash N \hookrightarrow_* Q : A$ and $\Delta \vdash P \hookrightarrow_* Q : A$.

Proof

By induction on the hypotheses.

We do not give here the complete proof of the lemma. Since the reduction rules are quite similar to those of [DL98], the proof is also similar to the proof of local confluence of [DL98]. The most interesting cases are those which involve the reduction rules of the *Case*/*It* operators. We have to check that the side conditions of these rules are still true after a reduction step. For instance, let us assume that $\mathrm{It}(I,Q,\uparrow\lambda\overrightarrow{z_i}:\overrightarrow{B_i}.(\mathrm{Constr}(I,k)\ M_1\ldots M_n))\{\overrightarrow{f_i}\}$ is well typed in Δ and that the variables z_i which appear in the dependent arguments of $\mathrm{Constr}(I,k)$ have a type that does not contain I. Moreover, let us assume that in Δ, $\lambda\overrightarrow{z_i}:\overrightarrow{B_i}.(\mathrm{Constr}(I,k)\ M_1\ldots M_n)$ reduces to $\lambda\overrightarrow{z_i}:\overrightarrow{B_i}.(\mathrm{Constr}(I,k)\ M'_1\ldots M_n)$. Then M'_1 cannot contain more variables z_i than M_1 and the side condition of the reduction rule is still valid.

□

4.2 Strong Normalization

We have not proved the strong normalization property for our system. This proof would require an important amount of work. Gilles Barthe has recently solved Ghani's conjecture [Bar99], i.e. he has proved strong normalization for CC with η-expansion. This is an important step towards the proof of strong normalization for our system.

Note that the proof of the strong normalization property for the LF sub-part of the system should be much easier. As for any λ-calculi in the cube, we can reduce the strong normalization problem for our dependent system to the strong normalization problem for the same system without dependent types. Forgetting the dependent types of the LF sub-part of our system gives us a simply typed system close to the simply typed version of our system. We do not elaborate more on this here.

Once the have proved the strong normalization property for the system, we get the confluence property for it since we have already proved local confluence (theorem 5).

Of course we are a bit unhappy to leave the proof of the strong normalization property for future work.

Remember, however, that the Church-Rosser property for LF with η-conversion was only given in 90 [Sal90], and that a definition of $\beta\eta$-long forms for the cube (hence for CC) was only given in 93 [DHW93], that is 3 to 8 years after these systems have been defined. (First presentations of CC appeared in 85 [CH85,CH88] while first presentations of LF appeared in 87 [HHP87,HHP93]). These systems are very interesting, but their meta-theoretical studies are rather complex, expecially for η-expansion.

5 Conclusion and Future Work

We have presented a system allowing recursion on HOAS, with dependent types. We only provided a non dependent elimination rule for dependent types and only proved preservation of typing and local confluence for the system. However, to our knowledge, this system is the most achieved system reconciling HOAS and recursion, and the first one allowing dependent types.

Primitive recursion is a bit more than case, iteration and pairs in the general case, although it is well known that primitive recursion can be defined over the naturals using iteration and pairs, as we have done in [DPS97]. Actually, it seems that providing a primitive recursion construct here would give an induction principle for terms defined by means of HOAS.

Similarly, it appears that giving a full treatment (i.e. rules for strong eliminations) for dependent types would mean providing a solution to both the problems of recursion and induction.

Thus the next challenge is to add induction principles. There exist several proposals for this in the literature [MM97,PS98]. However, the systems proposed there are not a single type theory, but a system in two levels; the induction principles being defined on a meta-level, above the level to which the terms belong.

Several induction principles for HOAS have been proposed so far. As induction is not in the scope of the present paper, we will not present these induction principles here. Let us just present the current situation concerning them.

First, induction principles for higher-order terms such as the ones of the untyped λ-terms $term$ and $term_v$ recalled in the beginning of the present paper (in section 1 and 2.1) have been commonly used in the LF and λ-prolog community for a long time.

Then, in [DH94,DFH95], we presented several induction principles on *higher-order terms* $term_{v,n}$, where we introduced these terms.

Our latter joint work [DL98] started from the idea that we should be able to deal with the "right" terms ($term_n$) and get the corresponding induction principle on this type, provided we add boxes, as we did in [DPS96,DPS97]. In our mind, this principle should have been the same as before, for $term_n$, equipped with boxes in the natural way.

Martin Hofmann [Hof99] recently proved the correctness of (closed forms of) both the induction principle commonly used in the LF community (for $term_v$) and the last one we mentionned (for $term_{v,n}$ with boxes). The induction principles he proposed differ from ours in that they do not include the cases for variables or projections. Martin Hofman achieved this result by giving categorical interpretations of terms in $term_{v,n}$. This semantical characterisation should enable one to find a type theory for HOAS, with both primitive recursion and induction, with dependent types.

References

[Bar91] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, D. M. Gabbai, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991.

[Bar99] Gille Barthe. Expanding the cube. In *To appear in the proceedings of FOSSACS'99*, Springer-Verlag LNCS 664, 1999.

[CH85] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. *Lecture Notes in Computer Science*, 203, 1985.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani and G. Plotkin, editors, *Proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902, pages 124–138. Springer-Verlag LNCS, April 1995. Also appears as INRIA Research Report RR-2556.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order syntax and induction in Coq. In F. Pfenning, editor, *Proceedings of the fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR 94)*, volume 822, pages 159–173. Springer-Verlag LNAI, July 1994. Also appears as INRIA Research Report RR-2292 (June 1994).

[DHW93] G. Dowek, G. Huet, and B. Werner. On the definition of the eta-long normal form in type systems of the cube. Informal proceedings of the workshop "Types", Nijmegen, 1993.

[DL98] Joëlle Despeyroux and Pierre Leleu. A modal λ-calcul with iteration and case constructs. In *Proc. of the annual Types for Proofs and Programs seminar*, Springer-Verlag LNCS, March 1998.

[DL99] Joëlle Despeyroux and Pierre Leleu. Metatheoretic results for a modal λ-calcul. Journal version. Submitted for publication in 98-99, 1999.

[DPS96] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1996.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote and J. Roger Hindley, editors, *Proceedings of the TLCA 97 Int. Conference on Typed Lambda Calculi and Applications, Nancy, France, April 2–4*, pages 147–163. Springer-Verlag LNCS 1210, April 1997. An extended version is available as CMU Technical Report CMU-CS-96-172.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In IEEE, editor, *Proceedings of the International Conference on Logic In Computer Sciences, LICS*, 1999.

[Lel97] Pierre Leleu. A modal λ-calcul with iteration and case constructs. Research Report RR-3322, INRIA, December 1997.

[Lel98] Pierre Leleu. Metatheoretic results for a modal λ-calcul. Research Report RR-3361, INRIA, February 1998.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, Warsaw, Poland, June 1997. To appear.

[PM92] Christine Paulin-Mohring. Inductive definitions in the system coq. rules and properties. In J.F. Groote M. Bezem, editor, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, Springer-Verlag LNCS 664, pages 328–345, 1992. also available as a Research Report RR-92-49, Dec. 1992, ENS Lyon, France.

[PS98] F. Pfenning and C. Schürmann. Automated Theorem Proving in a Simple Meta Logic for LF. In *Proceedings of the CADE-15 Conference*, Lindau - Germany, July 1998.

[PW95] Frank Pfenning and Hao-Chi Wong. On a modal λ-calculus for S4. In S. Brookes and M. Main, editors, *Proceedings of the Eleventh Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, March 1995. *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.

[Sal90] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, France, May 1990.

[Wer94] B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris VII, Mai. 1994.