Under consideration for publication in Math. Struct. in Comp. Science

Recursion over Objects of Functional Type

Joëlle Despeyroux and Pierre Leleu †

INRIA–Sophia-Antipolis

Received May 2000

This paper presents an extension of the simply-typed λ-calculus allowing iteration and case reasoning over terms of functional types that arise when using higher order abstract syntax. This calculus aims at being the kernel for a type theory in which the user will be able to formalize logics or formal systems using the LF methodology, while taking advantage of new induction and recursion principles, extending the principles available in a calculus such as the Calculus of Inductive Constructions. The key idea of our system is the use of modal logic S4. We present here the system, its typing rules and reduction rules. The system enjoys the decidability of typability, soundness of typed reduction with respect to the typing rules, the Church-Rosser and Strong Normalization properties and it is a conservative extension over the simply-typed λ-calculus. These properties entail the preservation of the adequacy of encodings.

Contents

† This work was partly supported by the European ESPRIT project "Types for Proofs and Programs"

1. Introduction

Higher order abstract syntax (Harper et al., 1993) is a representation technique which proves to be useful when modelizing in a logical framework a language which involves bindings of variables. Thanks to this technique, the formalization of an (object-level) language does not need definitions for free or bound variables in a term. Nor does it need definitions of notions of substitutions, which are implemented using the meta-level application, i.e. the application available in the logical framework. Hypothetical judgments are also directly supported by the framework.

On the other hand, inductive definitions are frequent in mathematics and semantics of programming languages, and induction is an essential tool when developing proofs. Unfortunately it is well-known that a type defined by means of higher order abstract syntax cannot be defined as an inductive type in usual inductive type theories (like CCI (Paulin-Mohring, 1992; Werner, 1994), or Martin-Löf's Logical Framework for instance).

In a first step towards the resolution of this dilemma, Frank Pfenning, Carsten Schürmann and the first author have presented (Despeyroux et al., 1997) an extension of the simply-typed λ-calculus with recursive constructs (operators for iteration and case reasoning), which enables the use of higher order abstract syntax in an inductive type. To achieve that, they use the operator $\Box$ of modal logic IS4 to distinguish the types $A \to B$ of the functional terms well-typed in the simply-typed λ-calculus from the types $\Box A \to B$ of the functional terms possibly containing recursive constructs.

In this paper, we present an alternative presentation of their system that we claim to be better in several aspects. We use the same mechanism as them to mix higher order abstract syntax and induction but our typing and reduction rules are quite different. Indeed there are several presentations of modal λ-calculus IS4 (Pfenning and Wong, 1995; Bierman and de Paiva, 1996; Davies and Pfenning, 1996). We have chosen the first variant (Pfenning and Wong, 1995), which has context stacks instead of simple contexts. This peculiarity creates some difficulties in the metatheoretical study but the terms generated by the syntax are simpler than those of (Despeyroux et al., 1997) (no 'let box' construction), and so this system is more comfortable to use. This choice probably makes our system more suitable for future extension with dependent types.

Moreover, instead of introducing an operational semantics which computes the canonical form (η-long normal form) using a given strategy, our system has reduction rules, which allow a certain nondeterminism in the mechanism of reduction. We have been able to adapt classic proof techniques to show the important metatheoretic results: decidability of typability, soundness of typing with respect to typing rules, Church-Rosser property (CR), Strong Normalization property (SN) and conservativity of our system with respect to the simply-typed λ-calculus. The main problems we encountered in the proofs are on one hand due to the use of functional types in the types of the recursive constructors, and on the other hand due to the use of η-expansion. To solve the problems due to η-expansion, we benefit from previous works done for the simply-typed λ-calculus (Jay and Ghani, 1995) and for system F (Ghani, 1996).

The remainder of this paper is organized as follows: Section 2 reviews the idea of higher-order abstract syntax and the use of modal operators to reconcile this method

with induction. We give simple examples of iteration and case analysis in Section 3. We introduce our system in section 4: its syntax, its typing and reduction rules. Then, in section 5, we prove its essential properties (soundness of typing, CR, SN) from which we deduce that it is a conservative extension of the simply-typed λ-calculus. Finally, we discuss related works and outline future work.

This paper is an extended and polished version of (Despeyroux and Leleu, 1998). A full version with complete technical developments is available in (Leleu, 1997).

2. Higher-Order Abstract Syntax

Higher-Order Abstract Syntax (HOAS)

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.

Let us introduce here a simple example of representation using HOAS, that will be useful later when we illustrate the mechanism of the reduction rules.

Suppose we want to represent the untyped λ-terms in the simplest Logical Framework we can think of: the simply-typed λ-calculus. We introduce the type L of untyped λ-terms together with two constructors $\texttt{lam} : (\texttt{L} \to \texttt{L}) \to \texttt{L}$ and $\texttt{app} : \texttt{L} \to \texttt{L} \to \texttt{L}$.

It is well-known (Harper et al., 1993) that the canonical forms (β-normal η-long) of type L are in one-to-one correspondence with the closed untyped λ-terms and that this correspondence is compositional. For instance the term $(\texttt{lam}\ \lambda\texttt{x} : \texttt{L}.(\texttt{app x x}))$ of type L represents the untyped λ-term $\lambda x.(x\ \ x)$.

Higher-Order Abstract Syntax and Induction, Modal Operators

The constructors `lam` and `app` above do not define an inductive type in usual inductive type theories like the Calculus of Inductive Constructions (Paulin-Mohring, 1992; Werner, 1994) or the Extended Calculus of Constructions because of the leftmost occurrence of L in the type of constructor `lam`. If we allowed this kind of inductive definition, we would be confronted with two serious problems. First, we would lose the one-to-one correspondence between the objects we represent and the canonical forms of type $L \to \cdots \to L$. For instance, if we have a Case construct (definition of a function by case over inductive terms), the term $(\texttt{lam}\ \lambda\texttt{x} : \texttt{L.Case x of}\ \ldots)$ does not represent any untyped λ-term. Moreover we would lose the strong normalization property; more precisely we could write terms which would reduce to themselves. Our goal is to introduce a system which repairs these deficiencies.

Following (Despeyroux et al., 1997), we shall use the modal operator $\Box$ of modal logic IS4 to distinguish the types $A \to B$ only containing *parametric functions* (functions built only from variables and the constructors of the types representing the object-level terms - (`lam`) and (`app`) for terms in L) from the types $\Box A \to B$ denoting the full function space containing recursive functions. For instance, in our system, a term such as the one above will be written $\lambda\texttt{x} : \Box\texttt{L.Case x of}\ \ldots$, of type $\Box L \to L$ whereas constructor `lam`

will have type $(L \to L) \to L$. Thus, our typing judgment will rule out undesirable terms such as (`lam λx : L.Case x of ...`).

Intuitively, the type $\Box A$ denotes the type of *closed terms*. We can iterate or perform case analysis on closed terms because all constructors are statically known. This is not the case if an object may contain some unknown free variables. A useful idea here is to represent open terms depending on n variables as closed terms (in the usual sense), that we called *higher-order terms* in previous work.

Higher-Order Abstract Syntax Revisited

In (Despeyroux and Hirschowitz, 1994; Despeyroux et al., 1995), we introduced what we called the *higher-order terms*, which are functions from an arbitrary (or a fixed) number of variables to terms. To recall the definition of those higher-order terms, let us consider as an example the untyped λ-terms built on a predefined set of variables:

$$\begin{array}{l} \mathrm{var}_v : \mathrm{var} \to L_{\{v\}} \\ \mathrm{app}_v : L_{\{v\}} \to L_{\{v\}} \to L_{\{v\}} \\ \mathrm{lam}_v : (\mathrm{var} \to L_{\{v\}}) \to L_{\{v\}} \end{array}$$

The higher-order terms corresponding to terms in $L_{\{v\}}$, belong to the type:

$$L_{\{v,n\}} = var \to \cdots \to var \to L.$$

Our later meta-theoretical studies of various systems (Despeyroux and Leleu, 1998; Despeyroux and Leleu, 1999) make extensive use of the following variant of $L_{\{v,n\}}$:

$$L_{\{n\}} = L \to \cdots \to L \to L.$$

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

$$\begin{array}{l} \mathcal{R}ef = \lambda n : nat.\lambda i \in [0..n-1].\lambda x_{n-1}, \ldots, x_0 : L.\ x_i \\ \mathcal{A}pp = \lambda n : nat.\lambda e, e' : L_{\{n\}}.\lambda x_{n-1}, \ldots, x_0 : L.\ (app\ (e\ x_{n-1}\ \ldots\ x_0)\ (e'\ x_{n-1}\ \ldots\ x_0)) \\ \mathcal{L}am = \lambda n : nat.\lambda e : L_{\{n+1\}}.\lambda x_{n-1}, \ldots, x_0 : L.\ (lam\ \lambda y : L.\ (e\ y\ x_{n-1}\ \ldots\ x_0)). \end{array}$$

Thus, in our view of HOAS, an untyped λ-term with n free variables in L is represented at the meta level by a term of type $L_{\{n\}}$. We use vectorial notations $\overrightarrow{x} : \overrightarrow{L}$ for $x_1 : L, \ldots, x_n : L$. Untyped λ-terms will be represented by terms of one of the following forms:

$$\begin{array}{l} \mathcal{A}pp_n(M,N) = \lambda \overrightarrow{x} : \overrightarrow{\mathrm{L}}.(\mathrm{app}\ (M\ \overrightarrow{x})\ (N\ \overrightarrow{x})) \\ \mathcal{L}am_n(M) = \lambda \overrightarrow{x} : \overrightarrow{\mathrm{L}}.(\mathrm{lam}\ (M\ \overrightarrow{x})) \\ \mathcal{R}ef_n(i) = \lambda \overrightarrow{x} : \overrightarrow{\mathrm{L}}.x_i \end{array}$$

These terms involve the higher-order constructors introduced above:

$$\begin{array}{l} \mathcal{R}ef_n : [0 \ldots n-1] \to L_{\{n\}} \\ \mathcal{A}pp_n : L_{\{n\}} \to L_{\{n\}} \to L_{\{n\}} \\ \mathcal{L}am_n : L_{\{n+1\}} \to L_{\{n\}} \end{array}$$

3. Examples

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

Informally, addition can be described by the following rules:

$$\text{plus } 0\ n = n \qquad \text{plus } (s\ m)\ n = s\ (\text{plus } m\ n)$$

In our system, we may view iteration as replacing constructors of inductive types by functions of appropriate type, as done in (Despeyroux et al., 1997). 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.

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

$$plus = \lambda m : \Box nat.\lambda n : \Box nat.(It\ m\ of\ M_0\ M_s)$$
$$\text{where}\quad M_0 = n \qquad M_s = \lambda x.\Box nat.\uparrow (s\ \downarrow x)$$

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

Example 2. Function Count

We can informally define the function which counts the number of occurences of bound variables in an untyped λ-term by:

$$\text{Count}(\text{app } M\ N) = \text{Count}(M) + \text{Count}(N)$$
$$\text{Count}(\text{lam } \lambda x : L.(M\ x)) = \text{Count}(M\ x),\ \text{where Count}(x) = 1$$

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

$$\text{Count} := \lambda M : \Box L.\ (\text{It } M \text{ of } M_{\text{app}}\ M_{\text{lam}}\ ) : \Box L \to \Box nat$$
$$\text{with}\quad M_{\text{app}} = \lambda m, n : \Box nat.\ (\text{plus } m\ n) \qquad M_{\text{lam}} = \lambda p : \Box nat \to \Box nat.(p\ \uparrow (S\ 0)))$$

Example 3. Function Form

A function giving the form of an untyped λ-term can be informally defined as follows:

$$\text{Form}(\lambda x.x) = 0 \qquad \text{Form}(\lambda x.(M\ x)\ (N\ x)) = 1 \qquad \text{Form}(\lambda x.(M\ x)) = 2$$

Note that, as in previous work (Despeyroux et al., 1997), 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 $\Box(L \to L) \to nat$, which will consist of a function

(the case analysis) applied to 0 - the value for variables:

$$\text{Form} = \lambda M : \Box(L \to L).\ ((\text{Case } M \text{ of } M_{\text{app}}\ M_{\text{lam}})\ \ 0) : \Box(L \to L) \to nat$$
$$\text{with}\quad M_{\text{app}} = \lambda x : \Box(L \to L).\lambda y : \Box(L \to L).2 \qquad M_{\text{lam}} = \lambda f : \Box(L \to L \to L).1$$

4. The System

In this section, we present the syntax, the typing rules and the semantics of our system.

4.1. *Syntax*

The system we present here is roughly the simply-typed λ-calculus extended by pairs, modality IS4 and recursion. We discuss the addition of polymorphism and dependent types in the conclusion.

Types To describe the types of the system, we consider a countable collection of constant types L_j ($j \in \mathbb{N}$), called the *ground* types. In our approach, they play the role of *inductive* types. The *types* are inductively defined by:

$$\text{Types} : T := L_j \mid T_1 \to T_2 \mid T_1 \times T_2 \mid \Box T$$

A type is said to be *pure* if it contains no $\Box$ operator and no product.

Context stacks Following the presentation of (Pfenning and Wong, 1995), we have *context stacks* instead of simple contexts. As usual a context Γ is defined as a list of unordered declarations $x : A$ where all the variables are distinct. A *context stack* Δ is an ordered list of contexts, separated by semi colons $\Gamma_1; \ldots; \Gamma_n$. "." denotes the empty context as well as the empty stack. Each stack can be viewed as a Kripke world.
Notations. A context stack is said to be *valid* if all the variables of the stack are distinct. We call *local* context of a stack $\Delta = \Gamma_1; \ldots; \Gamma_n$ the last context of the stack: Γ_n. The notation Δ, Γ, where Δ is a stack $\Gamma_1; \ldots; \Gamma_n$ and Γ is a context, is the stack $\Gamma_1; \ldots; \Gamma_n, \Gamma$. Similarly, the notation Δ, Δ', where Δ is the stack $\Gamma_1; \ldots; \Gamma_n$ and Δ' is the stack $\Gamma'_1; \ldots; \Gamma'_m$, is the stack $\Gamma_1; \ldots; \Gamma_n, \Gamma'_1; \ldots; \Gamma'_m$. If Δ is a valid stack of m contexts $\Gamma_1; \ldots; \Gamma_m$ and n is an integer, Δ^n denotes the stack Δ where the last n contexts have been removed: $\Gamma_1; \ldots; \Gamma_{m-n}$ if $n < m$, and the empty stack "." if $n \geq m$.

Terms We have seen in section 2 that we view open terms of type L, depending on n variables of type L, as *functional terms* of type $L_{\{n\}} = L \to \cdots \to L \to L$. For example, terms in the untyped λ-calculus given in section 2 will have three possible forms:

$$\mathcal{A}pp_n(M, N) = \lambda \overrightarrow{x} : \overrightarrow{\mathrm{L}}.(\text{app } (M\ \overrightarrow{x})\ (N\ \overrightarrow{x}))$$
$$\mathcal{L}am_n(M) = \lambda \overrightarrow{x} : \overrightarrow{\mathrm{L}}.(\text{lam } (M\ \overrightarrow{x}))$$
$$\mathcal{R}ef_n(i) = \lambda \overrightarrow{x} : \overrightarrow{\mathrm{L}}.x_i$$

In general the type of a constructor of a pure type L contains other types than L. Before describing the set of the terms, we consider a finite collection of constant terms

(the *constructors*) $C_{j,k}$, given with their pure type: $\left(B_{j,k,1} \to \cdots \to B_{j,k,n_{j,k}}\right) \to L_j$, where each $B_{j,k,l}$ is a pure type and L_j is a ground type. If $n_{j,k} = 0$, the type of $C_{j,k}$ is simply L_j.

The *terms* are inductively defined by:

$$\begin{aligned}\text{Terms} : M \quad := \quad & x \mid C_{j,k} \mid (M\ N) \mid \lambda x : A.M \mid \uparrow M \mid \downarrow M \mid \langle M_1, M_2\rangle \\ & \mid \text{fst } M \mid \text{snd } M \mid \langle\sigma\rangle\text{Case } M \text{ of } (M_{j,k}) \mid \langle\sigma\rangle\text{It } M \text{ of } (M_{j,k})\end{aligned}$$

where σ is a function mapping the ground types $L_j (j \in \mathbb{N})$ to types, and $(M_{j,k})$ are collections of terms indexed by the indexes of the constructors. The modal operator $\uparrow$ introduces an object of type $\Box A$ while the operator $\downarrow$ marks the elimination of a term of type $\Box A$. As usual, terms equivalent under α-conversion are identified.

4.2. *Typing and Reduction Rules on a Simple Example*

We give first the rules for case and iteration for the untyped λ-calculus example.

4.2.1. *Reduction Rules on untyped λ-terms*

We give first the reduction rules, which might help to understand the typing rules. For the sake of simplicity we introduce some notations.

Notations. We define two macros 'case' and 'it' by:

$$\begin{aligned}\text{case } M \quad &:= \quad \langle\sigma\rangle\text{Case } M \text{ of } M_{\text{app}}\ M_{\text{lam}} \\ \text{it } M \quad &:= \quad \langle\sigma\rangle\text{It } M \text{ of } M_{\text{app}}\ M_{\text{lam}}\end{aligned}$$

In our example, the reduction rules for case and iteration are the following ones:

$$\begin{aligned}(\text{case } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.(\text{app } P\ Q)) &\hookrightarrow \lambda\overrightarrow{u} : \overrightarrow{A}.(M_{\text{app}}\ \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.P\ \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.Q) \\ (\text{case } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.(\text{lam } P)) &\hookrightarrow \lambda\overrightarrow{u} : \overrightarrow{A}.(M_{\text{lam}}\ \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.P) \\ (\text{case } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.x_i) &\hookrightarrow \lambda\overrightarrow{u} : \overrightarrow{A}.u_i \\ \\ (\text{it } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.(\text{app } P\ Q)) &\hookrightarrow \lambda\overrightarrow{u} : \overrightarrow{A}.(M_{\text{app}}\ ((\text{it } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.P)\ \overrightarrow{u}) \\ &\qquad\qquad ((\text{it } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.Q)\ \overrightarrow{u})) \\ (\text{it } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.(\text{lam } P)) &\hookrightarrow \lambda\overrightarrow{u} : \overrightarrow{A}.(M_{\text{lam}}\ ((\text{it } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.P)\ \overrightarrow{u})) \\ (\text{it } \uparrow\lambda\overrightarrow{x} : \overrightarrow{L}.x_i) &\hookrightarrow \lambda\overrightarrow{u} : \overrightarrow{A}.u_i\end{aligned}$$

The first argument of the Case and It constructs, M, is the inductive term to analyze (representing an untyped λ-term in our example). Remember that M has type $L_{\{n\}}$. The second one, M_{app}, of type $\Box L_{\{n\}} \to \Box L_{\{n\}} \to A$, is the function which processes the case of constructor "app". The third argument, M_{lam}, of type $\Box L_{\{n+1\}} \to A$, is the function which processes the case of constructor "lam". Roughly speaking the "Case" construct computes its result by applying M_{app} or M_{lam} to the sons of its main argument. For iteration, the mechanism of reduction is a bit different: the terms M_{app} of type $A \to A \to A$ and M_{lam} of type $(A \to A) \to A$ are applied to the result of iteration on the sons of the main argument. Operationally, the effect of iteration on a term M amounts

to replacing the constructors lam and app by the terms M_{lam} and M_{app} in M (as in (Despeyroux et al., 1997)).

Now since we want to benefit from higher order declarations, the main argument of Case/It may have a functional type. In particular we also want to be able to compute Case/It of a projection $\lambda\overrightarrow{x} : \overrightarrow{L}.x_i$ without a leftmost constructor. That is the reason for the functional type of Case/It constructs : they take as input the values of the computation for the projections (as in (Despeyroux et al., 1997)).

4.2.2. *Typing Rules on untyped λ-terms*

Except for the use of the $\Box$ operator, and the use of $L_{\{n\}}$ instead of L, they are pretty standard for the app case. Note how the use of $L_{\{n\}}$ enables us to extend the usual case (app) to the functional case (lam) in an intuitive manner:

$$\frac{\Delta;\Gamma \vdash M : \Box L_{\{n\}} \quad \Delta;\Gamma \vdash M_{\text{app}} : \Box L_{\{n\}} \to \Box L_{\{n\}} \to A \quad \Delta;\Gamma \vdash M_{\text{lam}} : \Box L_{\{n+1\}} \to A}{\Delta;\Gamma \vdash \langle\sigma\rangle\text{Case } M \text{ of } M_{\text{app}}\ M_{\text{lam}} : A_{\{n\}}}$$

$$\frac{\Delta;\Gamma \vdash M : \Box L_{\{n\}} \qquad \Delta;\Gamma \vdash M_{\text{app}} : A \to A \to A \qquad \Delta;\Gamma \vdash M_{\text{lam}} : (A \to A) \to A}{\Delta;\Gamma \vdash \langle\sigma\rangle\text{It } M \text{ of } M_{\text{app}}\ M_{\text{lam}} : A_{\{n\}}}$$

where $A = \sigma(L)$ is the resulting type of the case or iteration process on M.

The case and iteration functions take as arguments the resulting values for the n variables of the term M being analysed; hence the resulting type A_n for both operators in the above rules.

4.2.3. *Examples*

Let us recall here the Count example from section 3:

$\text{Count} := \lambda M : \Box L.\ (\text{It } M \text{ of } M_{\text{app}}\ M_{\text{lam}}\ ) : \Box L \to \Box nat$
with $M_{\text{app}} = \lambda m, n : \Box nat.\ (\text{plus } m\ n)$ and $M_{\text{lam}} = \lambda p : \Box nat \to \Box nat.(p\ \uparrow (S\ 0)))$

The number of occurences of bound variables in $(\text{lam } \lambda x : L.x)$ is $\uparrow (S\ 0)$ because:

$(\text{Count} \quad \uparrow \lambda x : L.x) \hookrightarrow \lambda u : \Box nat.u$ by the reduction rules for variables (1)

$(\text{Count} \quad \uparrow (\text{lam } \lambda x : L.x))$
$\hookrightarrow \quad (M_{\text{lam}} \quad (\text{Count} \quad \uparrow \lambda x : L.x))$ by the reduction rules for lam
$\hookrightarrow \quad (M_{\text{lam}} \quad \lambda u : \Box nat.u)$ by (1)
$\hookrightarrow_\beta \quad (\lambda u : \Box nat.u \quad \uparrow (S\ 0))$
$\hookrightarrow_\beta \quad \uparrow (S\ 0)$

To illustrate the mechanism of case analysis let us recall the example of the "Form" function given in section 3:

$$\begin{aligned}\text{Form } M := (\langle\sigma\rangle\text{Case } \ M \ \text{ of } \ & \lambda u : \Box(L \to L).\lambda v : \Box(L \to L).2 \\ & \lambda f : \Box(L \to L \to L).1 \quad 0)\end{aligned}$$

The function "Form" is of type $\Box(L \to L) \to nat$. Let us call M_{lam} the term λf :

$\Box(L \to L \to L).1$. The following reductions, for example, hold:

$$\begin{array}{l}(\text{Form } \uparrow \lambda x : L.x) \\ \hookrightarrow\ (\lambda n : nat.n\ 0) \ \text{ by the reduction rules for variables} \\ \hookrightarrow_\beta\ 0\end{array}$$

$$\begin{array}{l}(\text{Form } \uparrow \lambda x : L.(\text{lam } \lambda y : L.x)) \\ \hookrightarrow\ (\lambda n : nat.\ (M_{\text{lam}}\ \uparrow \lambda x : L.\lambda y : L.x)\ 0) \ \text{ by the reduction rules for lam} \\ \hookrightarrow_\beta\ (\lambda n : nat.1\ 0) \\ \hookrightarrow_\beta\ 1\end{array}$$

4.3. *Typing Rules*

The typing rules are a combination of the rules for simply-typed λ-calculus, for pairs and projections, for modal λ-calculus IS4 (Pfenning and Wong, 1995) and the new rules for the recursive constructs "Case" and "It". Due to lack of place we do not give the rules for pairs and projections here. The rules are written in Figure 1 and 2 with the following notations:

Notations $B_{j,k,1}, \ldots, B_{j,k,n_{j,k}}$ are pure types. L_j is an inductive type. $(T_i)_{i=1,\ldots,p}$ is a collection, possibly empty, of pure types. Each T_i can be decomposed as ${T_i}^1 \to \cdots \to {T_i}^{r_i} \to L_i$, where L_i is a ground type and each ${T_i}^j$ is a pure type. Given the types C, $D_1, \ldots, D_p$, we denote $D_1 \to \cdots \to D_p \to C$ by $\prod_{i=1}^{i=p} D_i.C$. The map σ from ground types to types is extended over pure types by the equation: $\sigma(A \to B) = \sigma(A) \to \sigma(B)$.

In the rule for case reasoning, each $M_{j,k}$ is a function which, in the reduction process, is applied to each son of the term M, abstracted over all the variables on which M depends. Similarly, a function defined by case analysis will eventually be applied to its results on each variable on which M depends. For each such variable of type T_i in M, this result must itself be closed, i.e. abstracted over all variables in the context. Each of these variables has type $T_1 \to \cdots \to T_p \to T_i^j$ for $j = 1 \cdots r_i$, as shown in the reduction rules in section 4.5. Thus the type of each result is ${T'}_i$ defined by:

$${T'}_i := \Box(T_1 \to \cdots \to T_p \to T_i^1) \to \cdots \to \Box(T_1 \to \cdots \to T_p \to T_i^{r_i}) \to \sigma(L_i).$$

In the above example of the untyped λ-terms (section 4.2.2), T_i is L and ${T'}_i$ is simply $\sigma(L)$, for all i.

$$\text{(Var)}\ \frac{x : A \in \text{ local context of } \Delta}{\Delta \vdash x : A}\ \Delta \text{ valid} \qquad (\lambda)\ \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x : A.M\ :\ A \to B} \qquad \text{(App)}\ \frac{\Delta \vdash M : A \to B \qquad \Delta \vdash N : A}{\Delta \vdash (M\ N) : B}$$

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

Fig. 1. Typing rules; Simple types and modality

$$(C_{j,k}) \qquad \Delta \vdash C_{j,k} : B_{j,k,1} \to \cdots \to B_{j,k,n_{j,k}} \to L_j \qquad \Delta \text{ valid}, \ n_{j,k} \in \mathbb{N}$$

$$(\text{Case}) \frac{\Delta \vdash M : \Box\left(\prod_{i=1}^{i=p} T_i.L_n\right) \qquad \Delta \vdash M_{j,k} : \prod_{q=1}^{q=n_{j,k}} \Box\left(\prod_{i=1}^{i=p} T_i.B_{j,k,q}\right).\sigma(L_j)}{\Delta \vdash \langle\sigma\rangle \text{Case } M \text{ of } (M_{j,k}) : \prod_{z=1}^{z=p} T'_z.\sigma(L_n)}$$

$$(\text{It}) \frac{\Delta \vdash M : \Box\left(\prod_{i=1}^{i=p} T_i.L_n\right) \qquad \Delta \vdash M_{j,k} : \prod_{q=1}^{q=n_{j,k}} \sigma(B_{j,k,q}).\sigma(L_j)}{\Delta \vdash \langle\sigma\rangle \text{It } M \text{ of } (M_{j,k}) : \prod_{i=1}^{i=p} \sigma(T_i).\sigma(L_n)}$$

Fig. 2. Typing rules for case and iteration

These typing rules may seem complex at first sight. They are a natural extension of the case of the untyped λ-terms given in section 4.2.2, keeping in mind that recursion is only allowed on closed terms.

Although expressed differently (with compact notations instead of algorithms), our typing rules are similar to those in (Despeyroux et al., 1997) (in which one can find many examples). Terms are the same (modulo some syntactic translation) and have the same types in both calculi. The two notable differences between the two systems are:

— As noticed before, the modal core is different. We have context stacks instead of two contexts and our modal rules ($\uparrow$), ($\downarrow$) and (Pop) are expressed in a very simple way (like in (Pfenning and Wong, 1995)).

— In our last two typing rules the terms $M_{j,k}$ are a priori indexed by all the indexes of the constructors. This means that we should define one term $M_{j,k}$ per constructor $C_{j,k}$. Actually, when computing over inductive terms of type L_n, we only need to define the terms $M_{j,k}$ such that L_n and L_j are "mutually" inductive. This notion is essential for a future implementation of an extension of this system as a logical framework but is quite orthogonal to the properties we state and we prove in the rest of this work. The interested reader will find the definition of mutual inductive types we could adopt here completely formalized in the full version of (Despeyroux et al., 1997).

4.4. *Basic Properties*

The system allows the same basic stack manipulations as the modal λ-calculus IS4 without operators for case and iteration (Pfenning and Wong, 1995). In particular, as usual, the typing judgments are preserved by thinning and strengthening. Later, these properties will still be true for typed reduction and the interpretations of types.

The substitution rule is still admissible:

Proposition 4.1 (Admissibility of (Subst) Rule).
The following rule is admissible:

$$(\text{Subst}) \frac{\Delta \vdash N : A \qquad \Delta, x : A \vdash M : B}{\Delta \vdash M[N/x] : B}$$

The inversion lemmas are not totally trivial because our typing rules are not syntax-driven. If we try to type a term of type $\Box A$, we can always apply rule Pop as well as the structural rule for M. Nevertheless, they remain fairly simple (see (Leleu, 1997)). A typically non standard inversion rule is the one for the App case. Rule App, seen from bottom to top, may break a modal type $\Box B$ into two types $A \to \Box B$ and A which are no more modal. Thus, when faced with an application, we try to apply the rule Pop as much as we can (i.e. remove the local context till we have free variables of the term in the local context).

Proposition 4.2 (Inversion lemma for App).
If $\Delta \vdash (M\ \ N) : B$ then there is a type A such that $\Delta^n \vdash M : A \to B$ and $\Delta^n \vdash N : A$ ($n \in \mathbb{N}$), where $n = 0$ if B is not of the form $\Box C$ and otherwise n is the least integer such that Δ^n contains some free variables of $(M\ \ N)$ in the local context.

The inversion lemmas allow us to prove "the uniqueness of type" property (i.e. if $\Delta \vdash M : A$ and $\Delta \vdash M : A'$ then $A = A'$) and to find an algorithm which determines if a term M is typable in a stack Δ and which returns the type of M in Δ if it is typable ("decidability of typing").

4.5. *Reduction Rules*

Now, we turn to the reduction rules of our system. They are inspired by the reduction rules for case and iteration that have been suggested to us by Martin Hofmann as a means to describe the evaluation mechanism of (Despeyroux et al., 1997). These reduction rules are also the ones underlying the terms and induction principles presented in (Despeyroux and Hirschowitz, 1994) in the context of the Calculus of Inductive Constructions. Indeed this research was undertaken with this main idea, and hope, in mind: our approach to HOAS (i.e. considering terms in $L_{\{n\}} = L \to \cdots \to L$ instead of terms of type L (Despeyroux and Hirschowitz, 1994)) should lead to a more natural system than the usual approach to HOAS (where semantics are given on terms which are not functional but whose subterms eventually are). We shall come back to this point in the conclusion of the paper.

Given a term of our calculus, what we want to obtain at the end of the computation is the term of the object language it represents. As we have seen earlier (section 2), the canonical forms (β-normal η-long) are in one-to-one correspondence with the object terms. Thus we want the computation to return canonical forms. That means our reduction rules will incorporate η-expansion.

The η-expansion reduction rule has been thoroughly studied (see (Cosmo and Kesner, 1993), (Akama, 1993), (Jay and Ghani, 1995)). Adopting it forces us to restrict the reduction rules in some way if we still want Strong Normalization. Thus the reduction we will consider will not be a congruence (more precisely it will not be compatible with the application) and this will induce slight changes in the usual schemes of the proofs of the Church-Rosser and Strong Normalization properties.

The purpose of these restrictions is to prevent η-expansions to create new β-redexes which generate reduction loops. For instance, if we allowed an abstraction to be η-

expanded, $\lambda x : A.M$ could reduce to $\lambda y : A.(\lambda x : A.M \ \ y)$ and then to $\lambda y : A.M[y/x]$, which is α-convertible to $\lambda x : A.M$. Thus we forbid η-expansion of an abstraction. Similarly, we cannot allow η-expansion of the left argument of an application because otherwise we would have $\Delta \vdash (M\ N) \hookrightarrow (\lambda z : A.(M\ z)\ N) \hookrightarrow (M\ N) : B$.

The choice of η-expansion also means we have to keep track of the types of the terms. Indeed a term can only be η-expanded if it has type $A \to B$. Thus we will define a notion of typed reduction.

The reduction relation is defined by the inference rules in Figures 3 (simple types and modality) and 4 (Case and It). We have omitted the product rules and the compatibility rules other than (App_l), which are straightforward.

$$(\beta)\ \frac{\Delta \vdash (\lambda x : A.P \ \ Q) : B}{\Delta \vdash (\lambda x : A.P \ \ Q) \hookrightarrow P[Q/x] : B} \qquad (\beta\Box)\ \frac{\Delta \vdash \downarrow\uparrow M : A}{\Delta \vdash \downarrow\uparrow M \hookrightarrow M : A}$$

$$(\eta)\ \frac{\Delta \vdash M : A \to B \quad M \text{ is not an abstraction} \quad x \text{ fresh}}{\Delta \vdash M \hookrightarrow \lambda x : A.(M\ x) : A \to B} \qquad (\eta\Box)\ \frac{\Delta \vdash \uparrow\downarrow M : \Box A}{\Delta \vdash \uparrow\downarrow M \hookrightarrow M : \Box A}$$

$$(\text{App}_\text{l})\ \frac{\Delta \vdash M \hookrightarrow M' : A \to B\ (\neq \eta\text{-step}) \qquad \Delta \vdash N : A}{\Delta \vdash (M \ \ N) \hookrightarrow (M' \ \ N) : B} \qquad (\text{Pop})\ \frac{\Delta \vdash M \hookrightarrow N : \Box A}{\Delta; \Gamma \vdash M \hookrightarrow N : \Box A}$$

Fig. 3. Reduction rules; Simple types and modality

As usual we define the relations $\hookrightarrow_*$ and $=$ (conversion) respectively as the reflexive, transitive and the reflexive, symmetric, transitive closures of $\hookrightarrow$.

Although we have not proved this, we believe that the normal forms defined here are the same (modulo some syntactic translation) than those defined in (Despeyroux et al., 1997). The difference lies in the expression of the rules. The first system uses term reduction while the second one uses evaluation rules (involving an external "elimination" process to replace constructors $C_{j,k}$ by terms $M_{n_{j,k}}$ in the case of iteration).

5. Metatheoretical Results

The classic properties of subject reduction, confluence and strong normalization have already been established for a modal λ-calculus IS4 without induction (Leleu, 1997; Despeyroux and Leleu, 2000). Here we extend these results to the recursive operators Case and It.

5.1. *First Results*

First, we state soundness of typed reduction with respect to typing rules. It is easily proved by induction on the derivation of the first hypothesis.

Theorem 5.1 (Soundness of reduction).
If $\Delta \vdash M \hookrightarrow M' : A$ then $\Delta \vdash M : A$ and $\Delta \vdash M' : A$.

$$(\text{Case } C_{j,k})\ \frac{\Delta \vdash \langle\sigma\rangle\text{Case } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(C_{j,k}\ M_1 \dots M_{n_{j,k}})\ \text{ of }\ (M_{j,k}) : \prod_{z=1}^{z=p} T'_z.\sigma(L_n)}{\begin{array}{l}\Delta \vdash \langle\sigma\rangle\text{Case } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(C_{j,k}\ M_1 \dots M_{n_{j,k}})\ \text{ of }\ (M_{j,k}) \hookrightarrow \\ \quad \lambda\overrightarrow{u} : \overrightarrow{T'}.(M_{j,k}\ \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_1\ \dots\ \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_{n_{j,k}}) : \prod_{z=1}^{z=p} T'_z.\sigma(L_n)\end{array}}$$

$$(\text{Case } x_k)\ \frac{\Delta \vdash \langle\sigma\rangle\text{Case } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(x_k\ M_1 \dots M_{r_k})\ \text{ of }\ (M_{j,k}) : \prod_{z=1}^{z=p} T'_z.\sigma(L_n)}{\begin{array}{l}\Delta \vdash \langle\sigma\rangle\text{Case } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(x_k\ M_1 \dots M_{r_k})\ \text{ of }\ (M_{j,k}) \hookrightarrow \\ \quad \lambda\overrightarrow{u} : \overrightarrow{T'}.(u_k\ \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_1\ \dots\ \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_{r_k}) : \prod_{z=1}^{z=p} T'_z.\sigma(L_n)\end{array}}$$

$$(\text{It } C_{j,k})\ \frac{\Delta \vdash \langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(C_{j,k}\ M_1 \dots M_{n_{j,k}})\ \text{ of }\ (M_{j,k}) : \prod_{i=1}^{i=p} \sigma(T_i).\sigma(L_n)}{\begin{array}{l}\Delta \vdash \langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(C_{j,k}\ M_1 \dots M_{n_{j,k}})\ \text{of }\ (M_{j,k}) \hookrightarrow \\ \quad \lambda\overrightarrow{u} : \overrightarrow{\sigma(T)}.(M_{j,k}\ \ (\langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_1\ \text{ of }\ (M_{j,k})\ \ \overrightarrow{u})\ \dots \\ \qquad\qquad (\langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_{n_{j,k}}\ \text{ of }\ (M_{j,k})\ \ \overrightarrow{u})) : \prod_{i=1}^{i=p} \sigma(T_i).\sigma(L_n)\end{array}}$$

$$(\text{It } x_k)\ \frac{\Delta \vdash \langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(x_k\ M_1 \dots M_{r_k})\ \text{ of }\ (M_{j,k}) : \prod_{i=1}^{i=p} \sigma(T_i).\sigma(L_n)}{\begin{array}{l}\Delta \vdash \langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.(x_k\ M_1 \dots M_{r_k})\ \text{of }\ (M_{j,k}) \hookrightarrow \\ \quad \lambda\overrightarrow{u} : \overrightarrow{\sigma(T)}.(u_k\ \ (\langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_1\ \text{ of }\ (M_{j,k})\ \ \overrightarrow{u})\ \dots \\ \qquad\qquad (\langle\sigma\rangle\text{It } \uparrow \lambda\overrightarrow{x} : \overrightarrow{T}.M_{r_k}\ \text{ of }\ (M_{j,k})\ \ \overrightarrow{u})) : \prod_{i=1}^{i=p} \sigma(T_i).\sigma(L_n)\end{array}}$$

Fig. 4. Reduction rules for case and iteration

The relationship between substitution and typed reduction is not as easy as in the simply-typed λ-calculus. If $P \hookrightarrow_* P'$ and $Q \hookrightarrow_* Q'$ then we do not have any more $P[Q/x] \hookrightarrow_* P'[Q'/x]$ because of the side-conditions of reduction rules (η) and (App_l). Thus we only prove weak forms of the usual results. For instance, if $\Delta, x : A \vdash P : B$ and $\Delta \vdash Q \hookrightarrow Q' : A$, we only state that there is a term R such that $\Delta \vdash P[Q/x] \hookrightarrow_* R : B$ and $\Delta \vdash P[Q'/x] \hookrightarrow_* R : B$. Nevertheless, these results enable us to prove the local confluence property:

Lemma 5.2 (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 $\Delta \vdash P \hookrightarrow_* Q : A$.

5.2. *Strong Normalization*

Now we briefly sketch our proof of the Strong Normalization theorem for our system. The proof follows the idea of normalization proofs "à la Tait" and is inspired by (Werner, 1994) (for the inductive part) and (Ghani, 1996) (for the η-expansion part).

5.2.1. *Reducibility Candidates*

First we give a definition of the reducibility candidates (Girard et al., 1989) adapted to our setting. Let us call Λ the set of our terms, defined in section 4.1.

Definition 5.3 (Reducibility Candidates).
Given a type A, the *reducibility candidates* CR_A are sets $\mathcal{C}$ of pairs (Δ, M) satisfying the following properties:

CR1 $\forall (\Delta, M) \in \mathcal{C}$, M is strongly normalizing in Δ (i.e. there is no infinite sequence of reductions starting from M in Δ).
CR1' $\mathcal{C} \subset \{(\Delta, M) \mid \Delta \vdash M : A\}$
CR2 $\forall (\Delta, M) \in \mathcal{C}$ such that $\Delta \vdash M \hookrightarrow M' : A$, we have $(\Delta, M') \in \mathcal{C}$.
CR3 If $M \in \mathcal{NT}$, $\Delta \vdash M : A$ and forall M' such that $\Delta \vdash M \hookrightarrow M' : A$ ($\neq \eta$-expansion), $(\Delta, M') \in \mathcal{C}$ then we have $(\Delta, M) \in \mathcal{C}$.
CR4 If $A = B \to C$ and $(\Delta, M) \in \mathcal{C}$ then $(\Delta, \lambda z : B.(M\ \ z)) \in \mathcal{C}$, where z is a fresh variable.

where $\mathcal{NT} = \Lambda \setminus (\{\lambda x : A.M \mid M \in \Lambda\} \cup \{\uparrow M \mid M \in \Lambda\} \cup \{\langle M, N\rangle \mid M, N \in \Lambda\})$.

Note that instead of taking sets of terms, we consider sets of pairs of a stack and a term. Indeed, since, because of η-expansion, our reduction is typed, it is convenient for the reducibility candidates to contain well-typed terms. In rule CR3, the restriction "$\Delta \vdash M \hookrightarrow M' : A$ is not an η-expansion" comes from (Jay and Ghani, 1995). It has been introduced to cope with η-expansions. The rule CR4 is also needed because of the η-expansions (Ghani, 1996).

As usual, if $\mathcal{C}$ and $\mathcal{D}$ belong to CR_A then $\mathcal{C} \cap \mathcal{D}$ belong to CR_A. Thus CR_A is an *inf-semi lattice*. Next, we define the sets $\mathcal{C} \to \mathcal{D}$, $\mathcal{C} \times \mathcal{D}$, $\Box\mathcal{C}$ where $\mathcal{C}$ and $\mathcal{D}$ are two reducibility candidates:

Definition 5.4 ($\mathcal{C} \to \mathcal{D}$, $\Box\mathcal{C}$, $\mathcal{C} \times \mathcal{D}$).
— $\mathcal{C} \to \mathcal{D} := \{(\Delta, M) \mid \Delta \vdash M : A \to B$ and $\forall \Gamma, \forall ((\Delta, \Gamma), N) \in \mathcal{C}, ((\Delta, \Gamma), (M\ \ N)) \in \mathcal{D}\}$
— $\Box\mathcal{C} := \{(\Delta, M) \mid \Delta \vdash M : \Box A$ and $\forall \Delta'$ stack s.t. (Δ, Δ') is valid, $((\Delta, \Delta'), \downarrow M) \in \mathcal{C}\}$.
— $\mathcal{C} \times \mathcal{D} := \{(\Delta, M) \mid \Delta \vdash M : A \times B$ and $\forall \Gamma$ context s.t. (Δ, Γ) is valid, $((\Delta, \Gamma), \mathrm{fst}\ M) \in \mathcal{C}$ and $((\Delta, \Gamma), \mathrm{snd}\ M) \in \mathcal{D}\}$.

In the definition of $\Box\mathcal{C}$, we need to extend the stack of contexts Δ with Δ' in order to get $((\Delta, \Delta'), M) \in \Box\mathcal{C}$ whenever $(\Delta, M) \in \Box\mathcal{C}$ (similarly to the case of $\mathcal{C} \to \mathcal{D}$).

In the definition of $\mathcal{C} \to \mathcal{D}$, the context Γ added to the stack is essential; In the intermediate lemmas, it allows us to add fresh variables to the context.

Proposition 5.5. If $\mathcal{C}$ and $\mathcal{D}$ are two C.R., then $\mathcal{C} \to \mathcal{D}$, $\mathcal{C} \times \mathcal{D}$ and $\Box\mathcal{C}$ are C.R. too.

5.2.2. *Interpretation of Types and Contexts*
Following the sketch of normalization proofs "à la Tait", we define the interpretations of types.

Definition 5.6 (Interpretations of types).
— $[\![L_j]\!] := \{(\Delta, M) \mid \Delta \vdash M : L_j$ and M is SN in $\Delta\ \}$,
— $[\![A \to B]\!] := [\![A]\!] \to [\![B]\!]$,

— $[\![A \times B]\!] := [\![A]\!] \times [\![B]\!]$,
— If A is not pure, $[\![\Box A]\!] := \Box[\![A]\!]$

All the above interpretations are obviously C.Rs., except, maybe, for the first case:

Proposition 5.7 ($[\![L_j]\!]$ is a C.R.).
The set $[\![L_j]\!]$ is a reducibility candidate.

In order to define $[\![\Box A]\!]$ in the case A is pure, we have to take into account the fact that $\Box A$ may be the type of the inductive argument of Case/It. The definition of $[\![\Box A]\!]$ in this case involves the smallest fixpoint of a function which we do not give here, because of space limitation (see (Leleu, 1997)).

At this point, we have defined the interpretation of type $[\![A]\!]$ for all the types A. The following theorem stems from the definitions of the interpretations of types.

Theorem 5.8 ($[\![A]\!]$ is a C.R.).
Given any type A, the set $[\![A]\!]$ is a C.R.

Then we define the notion of *interpretation of context stack*. Like in the classic case of the simply-typed λ-calculus, the *interpretation* $[\![\Delta]\!]_\Psi$ of stack Δ in stack Ψ is a set of substitutions from Δ to Ψ but the definition is a bit more complex here because we have to deal with context stacks, instead of simple contexts. Thus we use a non standard notion of substitution.

Definition 5.9 (Pre-substitution).
A *pre-substitution* ρ from a stack Δ to a stack Ψ is a mapping from the set of the variables declared in Δ into the set of the terms with all their free variables in Ψ.

A *pre-substitution* ρ can be applied to a term M with all its free variables in Δ. The result of this operation, denoted by $\rho(M)$, is equal to term M where all its free variables x have been replaced by their images under ρ, $\rho(x)$.

Notations. Given two stacks Δ and Ψ, a pre-substitution ρ from Δ to Ψ, a variable x not declared in Δ and M a term with all its free variables in Ψ, we denote by $\rho[x \mapsto M]$ the pre-substitution from $\Delta, x : A$ to Ψ such that $\rho[x \mapsto M](y) = \rho(y)$ if y is declared in Δ and $\rho[x \mapsto M](x) = M$.

Given a stack Δ' such that $\Delta; \Delta'$ is valid and a substitution ρ' from Δ' to Ψ, $\rho; \rho'$ denotes the pre-substitution from $\Delta; \Delta'$ to Ψ such that $(\rho; \rho')(x) = \rho(x)$ if x is declared in Δ and $(\rho; \rho')(x) = \rho'(x)$ if x is declared in Δ'.

Definition 5.10 (Interpretation of context stack).
Given two stacks Δ and Ψ, the *interpretation* of Δ in Ψ, $[\![\Delta]\!]_\Psi$, is a set of pre-substitutions from Δ to Ψ. It is defined by induction on Δ:

— $[\![.]\!]_\Psi$ is the singleton whose only element is the empty pre-substitution from . to Ψ.
— $[\![\Gamma, x : A]\!]_\Psi$ is the set of the pre-substitutions $\rho[x \mapsto M]$, where ρ belongs to $[\![\Gamma]\!]_\Psi$ and (Ψ, M) is in $[\![A]\!]$.
— $[\![\Delta; \Gamma]\!]_\Psi$ is the set of pre-substitutions $\rho; \rho'$ such that ρ belongs to $[\![\Delta]\!]_{\Psi^n}$ $(n \in \mathbb{N})$ and ρ' belongs to $[\![\Gamma]\!]_\Psi$.

where the notation Ψ^n has been previously defined in section 4.1.

In the definition of $[\![\Delta;\Gamma]\!]_\Psi$, the requirement that ρ belongs to $[\![\Delta]\!]_{\Psi^n}$, which is more flexible than the requirement that ρ belongs to $[\![\Delta]\!]_\Psi$, enables us to cope with the context stacks in the proofs. For example, we will have that ρ belongs to $[\![\Delta;.]\!]_{\Psi,\Psi'}$ whenever ρ belongs to $[\![\Delta]\!]_\Psi$.

5.2.3. *Soundness of Typing*

This lemma is proved by induction on the derivation of $\Delta \vdash M : A$. The most difficult case occurs for rule ($\uparrow$). It is solved by using the typing restrictions imposed by modality (see (Leleu, 1997)).

Lemma 5.11 (Soundness of Typing).
If $\Delta \vdash M : A$ and $\rho \in [\![\Delta]\!]_\Psi$, then $(\Psi, \rho(M)) \in [\![A]\!]$.

The strong normalization theorem is then an easy corollary, using the fact that for any stack Δ, the pre-substitution identity from Δ to Δ belongs to $[\![\Delta]\!]_\Delta$.

Theorem 5.12 (Strong Normalization).
There is no infinite sequence of reductions.

5.3. *Confluence and Conservative Extension*

The *confluence* property is a corollary of the strong normalization (Theorem 5.12) and the local confluence results (this fact is often called "Newman's Lemma").

Theorem 5.13 (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 $\Delta \vdash P \hookrightarrow_* Q : A$.

As usual, the "uniqueness of normal forms" property is a corollary of the strong normalization and confluence theorems.

Corollary 5.14 (Uniqueness of normal forms).
If $\Delta \vdash M : A$ then M reduces to a unique canonical form in Δ.

The *conservative extension* property uses the strong normalization result together with a technical lemma, that defines the possible forms of a canonical term (Leleu, 1997).

Theorem 5.15 (Conservative extension).
Our system is a *conservative extension* of the simply-typed λ-calculus, i.e. if $\Delta \vdash M : A$ with Δ pure context stack and A pure type then M has a unique canonical form N which is *pure*.

6. Related Works

Our system has been inspired by (Despeyroux et al., 1997). The main difference is that the underlying modal λ-calculus is easier to use and seems to be better adapted to an extension to dependent types. Splitting the context in two parts (the intuitionistic and

the modal parts) would most probably make the treatment of dependent types even more difficult, when representing a modal type depending on both non-modal and modal types.

We also provide reduction rules, instead of a particular strategy for evaluation. Finally, due to that latter point and the fact that we have adapted well known proof methods, our metatheoretic proofs are much more compact and easier to read. On the whole, our system seems to be a better candidate for further extensions than the previous proposition.

Raymond McDowell and Dale Miller have proposed (McDowell and Miller, 1997) a meta-logic to reason about object logics coded using higher order abstract syntax. Their approach is quite different from ours, less ambitious in a sense. They do not provide a type system supporting the judgments-as-types principle. Instead, they propose two logics: one for each level (the object and meta levels). Moreover they only have induction on natural numbers, which can be used to derive other induction principles via the construction of an appropriate measure.

Frank Pfenning and Carsten Schürmann have also defined a meta-logic $\mathcal{M}_2$, which allows inductive reasoning over HOAS encodings in LF (Pfenning and Schürmann, 1998). Recent development of this meta-logic are a nice improvement of the system by Raymond McDowell and Dale Miller. It was designed to support automated theorem proving. This meta-logic has been implemented in the theorem prover Twelf, which gives a logical programming interpretation of $\mathcal{M}_2$. Twelf has been used to automatically prove properties such as type preservation for Mini-ML, an impressive result for us.

7. Conclusion and Future Work

We have presented a modal λ-calculus IS4 with primitive recursive constructs that we claim to be better than the previous proposition (Despeyroux et al., 1997). The conservative extension theorem, which guarantees that the adequacy of encodings is preserved, is proved as well as the Church-Rosser and strong normalization properties.

Dependent types. Our main goal is now to extend this system to dependent types and to polymorphic types. This kind of extension is not straightforward but we expect our system to be flexible enough to allow it. We have already proposed an extension of our system to dependent types, only with a "non-dependent" rule for elimination for the moment (Despeyroux and Leleu, 1999). 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 (Despeyroux et al., 1997). 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.

Induction. Thus the next challenge is to add induction principles. There exist several proposals for this in the literature (McDowell and Miller, 1997; Pfenning and Schürmann, 1998). 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 L and $L_{\{v\}}$ recalled in the beginning of the present paper (in section 2) have been commonly used in the LF and λ-prolog community for a long time.

Then, in (Despeyroux and Hirschowitz, 1994; Despeyroux et al., 1995), we presented several induction principles on *higher-order terms* in $L_{\{v,n\}}$. (These principles are derived by the Coq system from the various definitions of *valid* terms).

The work presented here started from the idea that we should be able to deal with higher-order terms in $L_{\{n\}}$ and get the corresponding recursion and induction principles on this type, provided we add boxes, as we did in (Despeyroux et al., 1997). In our mind, the induction principle should have been the same as before, adapted to $L_{\{n\}}$ equipped with boxes in the natural way.

Martin Hofmann (Hofmann, 1999) recently proved the correctness of (closed forms of) both the induction principle commonly used in the LF community (for $L_{\{v\}}$) and the last one we mentionned (for $L_{\{v,n\}}$ 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 $L_{\{v,n\}}$. This semantical characterisation should enable one to find a type theory for HOAS, with both primitive recursion and induction, with dependent types.

Another interesting direction of research consists in replacing our recursive operators by operators for pattern-matching such as those used in the ALF system, implementing Martin-Löf's Type Theory. Some hints for a concrete syntax for this extension have been given in (Despeyroux et al., 1997). Frank Pfenning and Carsten Schürmann are currently working on the definition of a meta-logic along these lines.

A related domain is the design of a programming language based on some features we have provided in our systems, in particular dependant types. Frank Pfenning and Hongwei Xi have recently proposed such a language, in the ML tradition (Pfenning and Xi, 1999).

Acknowledgments Thanks are due to Martin Hofmann for his suggestion for the initial form of the reduction rules which, stengthening us in the intuitions we had in previous works, make it possible the present results. We also thank André Hirschowitz for many fruitful discussions.

References

Akama, Y. (1993). On Mints' reduction for ccc-calculus. In *Proceedings TLCA*, pages 1–12. Springer-Verlag LNCS 664.

Bierman, G. and de Paiva, V. (1996). Intuitionistic necessity revisited. In *Technical Report CSRP-96-10*, School of Computer Science, University of Birmingham.

Cosmo, R. D. and Kesner, D. (1993). A Confluent Reduction for the Extensional Typed λ-calculus. In *Proceedings ICALP'93*. Springer-Verlag LNCS 700.

Davies, R. and Pfenning, F. (1996). A modal analysis of staged computation. In Guy Steele, J., editor, *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg Beach, Florida. ACM Press.

Despeyroux, J., Felty, A., and Hirschowitz, A. (1995). Higher-order abstract syntax in Coq. In Dezani, M. and Plotkin, G., editors, *proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902, pages 124–138. Springer-Verlag LNCS. Preliminary version available as INRIA Research Report RR-2556.

Despeyroux, J. and Hirschowitz, A. (1994). Higher-order syntax and induction in Coq. In Pfenning, F., editor, *proceedings of the fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR 94)*, volume 822, pages 159–173. Springer-Verlag LNAI. Preliminary version available as INRIA Research Report RR-2292 (June 1994).

Despeyroux, J. and Leleu, P. (1998). A modal λ-calcul with iteration and case constructs. In *proceedings of the annual Types for Proofs and Programs seminar*, Springer-Verlag LNCS 1657.

Despeyroux, J. and Leleu, P. (1999). Primitive recursion for higher-order abstract syntax with dependant types. In *Informal proceedings of the FLoC'99 IMLA Workshop on Intuitionistic Modal Logics and Applications.*

Despeyroux, J. and Leleu, P. (2000). Metatheoretic results for a modal lambda-calculus. *Journal of Functional and Logic Programming (JFLP)*, 2000(1).

Despeyroux, J., Pfenning, F., and Schrmann, C. (1997). Primitive recursion for higher-order abstract syntax. In de Groote, P. and Hindley, J. R., 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.

Ghani, N. (1996). Eta Expansions in System F. Technical Report LIENS-96-10, LIENS-DMI.

Girard, J.-Y., Lafont, Y., and Taylor, P. (1989). *Proofs and Types.* Cambridge University Press.

Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184.

Hofmann, M. (1999). Semantical analysis of higher-order abstract syntax. In IEEE Computer Society Press, editor, *Proceedings of the International Conference on Logic In Computer Sciences, LICS*, pages 204–213.

Jay, C. and Ghani, N. (1995). The Virtues of Eta-Expansion. *Journal of Functional Programming*, 5(2):135–154.

Leleu, P. (1997). A modal λ-calcul with iteration and case constructs. Research Report RR-3322, INRIA.

McDowell, R. and Miller, D. (1997). A logic for reasoning with higher-order abstract syntax: An extended abstract. In *Proc. of LICS'97.*

Paulin-Mohring, C. (1992). Inductive definitions in the system coq. rules and properties. In *Proc. of the TLCA'93 Int. Conference*, Springer-Verlag LNCS 664, pages 328–345.

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

Pfenning, F. and Wong, H.-C. (1995). On a modal λ-calculus for S4. In Brookes, S. and Main, M., editors, *Proceedings of the 11th MFPS Conference*, New Orleans, Louisiana. *Electronic Notes in TCS*, Volume 1, Elsevier.

Pfenning, F. and Xi, H. (1999). Dependant types in practical programming. In *Proceedings of the POPL'99 International Conference*, San Antonio, Texas.

Werner, B. (1994). *Une Thorie des Constructions Inductives.* PhD thesis, Universit Paris VII.