A Modal Lambda Calculus with Iteration and Case Constructs

Joëlle Despeyroux and Pierre Leleu

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

Abstract. 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, has recently been introduced by Joëlle Despeyroux, Frank Pfenning and Carsten Schürmann. This thorny mixing is achieved thanks to the help of the operator '$\Box$' of modal logic S4. Here we give a new presentation of their system, with reduction rules, instead of evaluation judgments, that compute the canonical forms of terms. Our presentation is based on a modal λ-calculus that is better from the user's point of view because it requires fewer annotations in terms. Moreover we do not impose a particular strategy of reduction during the computation. Our system enjoys the decidability of typability, soundness of typed reduction with respect to typing rules, the Church-Rosser and strong normalization properties and it is a conservative extension of the simply-typed λ-calculus.

1 Introduction

Higher order abstract syntax ([PE88]) is a representation technique which proves to be useful when modelizing in a logical framework a language which involves bindings of variables. By borrowing the variables of the meta level to represent the variables of the object language, one does not need to worry about variable binding, substitution or renaming. These operations are managed at the meta level. Parametric and 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 [Wer94], [PM92], or Martin-Löf's Logical Framework [NPS90] for instance).

In a first step towards the resolution of this dilemma, Joëlle Despeyroux, Frank Pfenning and Carsten Schürmann have presented ([DPS97]) 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 theirs 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 ([BdP96], [PW95], [DP96]). We have chosen the variant by Frank Pfenning and Hao-Chi Wong ([PW95]), 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 [DPS97] (no 'let box' construction), and so this system is more comfortable to use.

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 ([JG95]) and for system F ([Gha96]).

In the second section of the paper, we introduce our version of the modal inductive system, its syntax, its typing and reduction rules. Then in the third section, 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. A full version of this paper with complete technical developments is available in [Lel97].

2 The system

In this section, we present the syntax, the typing rules and the semantics of our system. First, let us briefly recall our motivations.

2.1 Higher-Order Abstract Syntax

The mechanics of higher order abstract syntax (HOAS) has already been exposed in many places, for example in [HHP93]. 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.

Let us assume that we work in the simply-typed λ-calculus with no extra equations, where we want to represent the untyped λ-terms. We introduce the type L of untyped λ-terms together with two constructors:

- $\texttt{lam} : (\texttt{L} \rightarrow \texttt{L}) \rightarrow \texttt{L}$
- $\texttt{app} : \texttt{L} \rightarrow \texttt{L} \rightarrow \texttt{L}$

It is well-known ([HHP93]) 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 of type L $(\texttt{lam}\ \lambda\texttt{x} : \texttt{L}.(\texttt{app x x}))$ represents the untyped λ-term $\lambda x.(x\ \ x)$.

Now, these constructors do not define an inductive type in usual inductive type theories like the Calculus of Inductive Constructions ([Wer94]) or the Extended Calculus of Constructions ([Luo94]) because of the leftmost occurence 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 $L \rightarrow \cdots \rightarrow L$. For instance, if we have a Case construct (definition of a function by case over inductive terms), the term $(\texttt{lam}\ \lambda\texttt{x} : \texttt{L}.\texttt{Case x of } \ldots)$ does not represent any untyped λ-term. Moreover we would lose the important property of strong normalization; more precisely we could write terms which would reduce to themselves. Our goal is to introduce a system which repairs these deficiencies.

Following [DPS97], we will use the modal operator '$\Box$' of modal logic IS4 to distinguish the types '$A \rightarrow B$' of the functional terms well-typed in the simply-typed λ-calculus from the types '$\Box A \rightarrow B$' of the functional terms possibly containing recursive contructs. For instance, in our system, a term such as '$\lambda x : L.\text{Case } x \text{ of } \ldots$' will have type $\Box L \rightarrow L$ whereas constructor 'lam' will have type $(L \rightarrow L) \rightarrow L$. Thus, our typing judgment will rule out undesirable terms such as '$(\text{lam}\ \ \lambda x : L.\text{Case } x \text{ of } \ldots)$'.

2.2 Syntax

The system we present here is roughly the simply-typed λ-calculus extended by pairs, modality IS4 and recursion. The addition of polymorphism and dependent types is left for future work.

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 [PW95], 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.
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 view open terms of type L, depending on n variables of type L, as *functional terms* of type $L_n = L \to \cdots \to L \to L$, as in [DH94]. For example, terms in the untyped λ-calculus given in Section 2.1 will have three possible forms, involving what we called *higher-order constructors*, written with vectorial notations:

$$\begin{array}{l} \text{app} : \lambda \overrightarrow{x} : \overrightarrow{L}.(\text{app } P\ Q) \\ \text{lam} : \lambda \overrightarrow{x} : \overrightarrow{L}.(\text{lam } P) \\ \text{var} : \ \lambda \overrightarrow{x} : \overrightarrow{L}.x_i \end{array}$$

In general of course, 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:

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

where σ is a function mapping the ground types $L_j (j \in \mathbb{N})$ to types, $(M_{j,k})$ and $({M'}_{j,k})$ are collections of terms indexed by the indexes of the constructors.

We delay till Section 2.6 the explanation of the arguments of operators 'Case' and 'It'. 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.

2.3 Typing Rules for Case and It on a simple example

We give here the typing rules for Case and It for the untyped λ-calculus example of Section 2.1. Except for the use of the $\Box$ operator, and the use of L_n instead of L, there are pretty standard for the app case. Note how the use of L_n enables us to extend the usual case (app) to the fonctional case (lam) in an intuitive manner:

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

$$\frac{\Delta;\Gamma \vdash M : \Box L_n \qquad \Delta;\Gamma \vdash M'_{\text{app}} : A \to A \to A \qquad \Delta;\Gamma \vdash M'_{\text{lam}} : (A \to A) \to A}{\Delta;\Gamma \vdash \langle\sigma\rangle\text{It } M \text{ of } M'_{\text{app}}\ M'_{\text{lam}} : A_n}$$

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

The Case and It 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.

2.4 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 ([PW95]) 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 in this extended abstract. The rules are written in Figure 1 with the following notations:

$$(\text{Var})\ \frac{x : A \in \text{ local context of } \Delta}{\Delta \vdash x : A}\ \Delta \text{ valid}$$

$$(\lambda)\ \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x : A.M \ :\ A \to B} \qquad (\text{App})\ \frac{\Delta \vdash M : A \to B \qquad \Delta \vdash N : A}{\Delta \vdash (M\ N) : B}$$

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

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

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

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

Fig. 1. Typing rules

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, ..., D_p, we denote $D_1 \to \cdots \to D_p \to C$ by $\prod_{i=1}^{i=p} D_i.C$.
We define T'_z by:

$$T'_z := \Box(T_1 \to \cdots \to T_p \to T_z^1) \to \cdots \to \Box(T_1 \to \cdots \to T_p \to T_z^{r_z}) \to \sigma(L_z)$$

The map σ from ground types to types is extended over pure types by the equation: $\sigma(A \to B) = \sigma(A) \to \sigma(B)$.

These typing rules may seem complex at first sight but they are naturally derived from the behaviour of the Case and It operators with respect to reduction (sections 2.6, 2.7).

Although expressed a bit differently, our typing rules are similar to those in [DPS96] (in which one can find many examples), except two notable differences:

- 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 [PW95]).
- In our last two typing rules the terms $M_{j,k}$ and $M'_{j,k}$ are a priori indexed by all the indexes of the constructors. This means that we should define one term $M_{j,k}$ (or $M'_{j,k}$) 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 [DPS96].

Note on the modal core of the system There exist several presentations of the modal core of our calculus in the literature. Let us briefly recall them here.

It is well known that the following rule for $\Box_I$, which would prevent the system to satisfy the substitution lemma, does not hold:

$$\frac{\Box\Gamma \vdash A}{\Box\Gamma \vdash \Box A}\ (\Box_I)$$

The introduction and elimination rules for the $\Box$ operator of the IS4 system presented in Natural Deduction syle, using contexts, are a bit more complicated ([BdP96]):

$$\frac{\Gamma \vdash \Box A_1 \quad \cdots \quad \Gamma \vdash \Box A_n \qquad \Box A_1, \cdots, \Box A_n \vdash \Box A}{\Gamma \vdash \Box A}\ (\Box_I) \qquad \frac{\Gamma \Box A}{\Gamma \vdash A}\ (\Box_E)$$

Adding terms to the above system in order to get a λ-calculus verifying the Curry-Howard isomorphism, Gavin Bierman and Valeria De Paiva obtained the following rules ([BdP96]):

$$\frac{\Gamma \vdash M_1 : \Box A_1 \cdots \Gamma \vdash M_n : \Box A_n \quad x_1 : \Box A_1 \ldots x_n : \Box A_n \vdash N : B}{\Gamma \vdash \text{box } N \text{ with } M_1, \ldots, M_n \text{ for } x_1, \ldots, x_n : \Box B}\ (\Box_I) \quad \frac{\Gamma \vdash M : \Box A}{\Gamma \vdash \text{unbox}(M) : A}\ (\Box_E)$$

Frank Pfenning and Rowan Davies found an interesting variant [DP96] of the previous system by splitting the contexts into two parts: one (Δ) for intuitionistic hypothesis, and a second one (Γ) for modal hypothesis. The introduction rule for $\Box$ becomes much simpler, while the elimination rule now needs a "let" construct to use modal hypothesis:

$$\frac{\Delta; . \vdash M : A}{\Delta; \Gamma \vdash \text{box } M : \Box A}\ (\Box_I) \qquad \frac{\Delta; \Gamma \vdash M : \Box A \qquad (\Delta, x : A); \Gamma \vdash N : B}{\Delta; \Gamma \vdash \text{let box } x = M \text{ in } N : B}\ (\Box_E)$$

Finally, Frank Pfenning and Hao-Chi Wong proposed an implicit variant of the above system, which no longer needs a let construct [PW95]. The idea is to use context stacks ($\Delta = \Gamma_1, \cdots, \Gamma_n$)

instead of usual contexts in the rules. Each context Γ_i maybe seen as a Kripke model. The price to pay, besides this non standard use of contexts, is the need for an extra "Pop" rule which makes the system apparently non deterministic (see rules $\uparrow$, $\downarrow$ and *Pop* in Figure 1).

We chose the lattest system because it is the simplest and more elegant one, as far as syntax is concerned. However, both the Pop rule and the non standard use of context stacks complicate a little bit the meta-theoretical study of the system.

The interested reader might find various mappings between the different presentations of the IS4 systems in the literature (see [DP96,PW95]).

2.5 Basic Properties

The system allows the same basic stack manipulations as the modal λ-calculus IS4 without Case and It ([PW95]). 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 1 (Admissibility of (Subst) Rule).
The following rule is admissible:

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

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 [Lel97]). 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 2 (Inversion lemmas for (App)).
If $\Delta \vdash (M \quad N) : B$ then there is a type A such that $\Delta^n \vdash M : A \to B$ and $\Delta^n \vdash N : A$ ($n \in I\!N$), 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 $(M \quad N)$.

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 typability").

2.6 Reduction rules for Case and It on a simple example

Now, we turn to the reduction rules of our system. They are inspired by the reduction rules for Case and It that have been suggested to us by Martin Hofmann as a means to describe the evaluation mechanism of [DPS97]. These reduction rules are also the ones underlying the terms and induction principles presented in [DH94] in the Calculus of Inductive Constructions. Indeed this research was undertaken with this main idea in mind: our approach to HOAS (i.e. considering terms in $L_n = L \to \cdots \to L$ instead of terms of type L ([DH94])) should lead to a much more elegant system than the usual approach. The result confirms our intuition.

First we show the reduction rules for Case and It in the simple setting of the example of Section 2.1. For the sake of simplicity we introduce some notations.

Notations. For any type B, the type B_n ($n \in \mathbb{N}$) is defined by $B_0 := B$ and $B_{n+1} := B \to B_n$. We consider a map σ from the inductive types to types such that $\sigma(L) = A$, terms M_{app} of type $\Box L_n \to \Box L_n \to A$, M_{lam} of type $\Box L_{n+1} \to A$, M'_{app} of type $A \to A \to A$ and M'_{lam} of type $(A \to A) \to A$. We define two macros 'case' and 'it' by:

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

Reduction rules. In our example, the reduction rules for Case and It are the following ones:

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

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

The first argument of the Case and It constructs, M, is the inductive term to analyze (representing an untyped λ-term in our example). The second one, M_{app}, is the function which processes the case of constructor 'app'. The third argument, M_{lam}, 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} and M'_{lam} are applied to the result of 'It' on the sons of the main argument. Operationally, the effect of 'It' on a term M amounts to replacing the constructors lam and app by the terms M'_{lam} and M'_{app} in M (see [DPS97]).

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 (see [DPS97]).

Examples. Let us assume that we have defined the types of the integers 'Nat' by declaring two constructors '0 : Nat' and 'S : Nat $\to$ Nat'. We can informally define the function which counts the number of bound variables in an untyped λ-term by:

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

This function can be implemented in our system using the It construct, where $\sigma = \{L \mapsto Nat\}$:

$$\text{Count} := (\langle\sigma\rangle\text{It } \; \lambda m : \Box\text{Nat}.\lambda n : \Box\text{Nat}.(\text{plus } m \; n) \; \; \lambda p : \Box\text{Nat} \to \Box\text{Nat}.(p \; \uparrow (S \; 0)))$$

Count has type $\Box L \to \Box\text{Nat}$. Let us give an example of computation: let us count the number of bound variables in the untyped λ-term $(\lambda x.x \;\; \lambda x.x)$:

$$\begin{aligned} &. \vdash (\text{Count } \; \uparrow (\text{app } (\text{lam } \lambda x : L.x) \; (\text{lam } \lambda x : L.x))) \hookrightarrow_* \\ &\quad (\lambda m : \Box\text{Nat}.\lambda n : \Box\text{Nat}.(\text{plus } m \; n) \; \; (\text{Count } \; \uparrow (\text{lam } \lambda x : L.x)) \; \; (\text{Count } \; \uparrow (\text{lam } \lambda x : L.x))) \end{aligned}$$

which reduces to $\uparrow (S \; (S \; 0))$ since:

$$\begin{aligned} &. \vdash (\text{Count } \; \uparrow (\text{lam } \lambda x : L.x)) \hookrightarrow (\lambda p : \Box\text{Nat} \to \Box\text{Nat}.(p \; \uparrow (S \; 0)) \; \; (\text{Count } \; \uparrow \lambda x : L.x)) \hookrightarrow_* \\ &\qquad ((\text{Count } \; \uparrow \lambda x : L.x) \; \; \uparrow (S \; 0)) \hookrightarrow (\lambda u : \Box\text{Nat}.u \; \; \uparrow (S \; 0)) \hookrightarrow \uparrow (S \; 0) \end{aligned}$$

To further illustrate the reduction mechanism, let us define the function 'Form' of type $\Box(L \to L) \to \mathrm{Nat}$, which returns 0 if its argument is a free variable, 1 if it is an abstraction term and 2 if it is an application.

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

For instance, the following reductions hold:

$$(\mathrm{Form}\ \ \uparrow \lambda x : L.x) \hookrightarrow (\lambda n : \mathrm{Nat}.n\ 0) \hookrightarrow 0$$

$$(\mathrm{Form}\ \ \uparrow \lambda x : L.(\mathrm{lam}\ \lambda y : L.x)) \hookrightarrow (\lambda n : \mathrm{Nat}.(\lambda f : \Box(L \to L \to L).1\ \uparrow \lambda x : L.\lambda y : L.x)\ 0) \hookrightarrow 1$$

2.7 Reduction rules

Now we describe the whole set of reduction rules. 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 2.1), 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 [CK93], [Aka93], [JG95]). 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 $\lambda y : A.(\lambda x : A.M\quad y)$ 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 2 (simple types and modality) and 3 (Case and It). We have omitted the product rules and the compatibility rules other than $(\mathrm{App_l})$, which are straightforward.

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

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

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

Fig. 2. 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 $\hookrightarrow$.

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

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

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

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

Fig. 3. Reduction rules for Case and It

3 Metatheoretical results

The classic properties of subject reduction, confluence and strong normalization have already been established for a modal λ-calculus IS4 without induction ([Lel98a]). Here we extend these results to the recursive operators Case and It.

3.1 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 3 (Soundness of reduction).
If $\Delta \vdash M \hookrightarrow M' : A$ then $\Delta \vdash M : A$ and $\Delta \vdash M' : A$.

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_1). 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 4 (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 $\Delta \vdash P \hookrightarrow_* Q : A$.

3.2 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 [Wer94] (for the inductive part) and [Gha96] (for the η-expansion part).

Reducibility Candidates

First we give a definition of the reducibility candidates ([GLT89]) adapted to our setting. Let us call Λ the set of our terms, defined in Section 2.2.

Definition 5 (Reducibility Candidates). Given a type A, the *reducibility candidates* CR_A are sets of pairs (Δ, M) of a context stack and a term. They are defined as follows:

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 $A = B \rightarrow C$ and $(\Delta, M) \in \mathcal{C}$ then $(\Delta, \lambda z : B.(M \quad z)) \in \mathcal{C}$, 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 [JG95]. It has been introduced to cope with η-expansions. The rule CR4 is also needed because of the η-expansions ([Gha96]).

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 $\mathcal{C} \rightarrow \mathcal{D}$, $\mathcal{C} \times \mathcal{D}$, $\Box\mathcal{C}$ where $\mathcal{C}$ and $\mathcal{D}$ are two reducibility candidates:

Definition 6 ($\mathcal{C} \rightarrow \mathcal{D}$, $\Box\mathcal{C}$, $\mathcal{C} \times \mathcal{D}$).

- $\mathcal{C} \rightarrow \mathcal{D} := \{(\Delta, M) \mid \Delta \vdash M : A \rightarrow B$ and $\forall\Gamma, \forall((\Delta, \Gamma), N) \in \mathcal{C}, ((\Delta, \Gamma), (M \quad N)) \in \mathcal{D}\}$
- $\Box\mathcal{C} := \{(\Delta, M) \mid \Delta \vdash M : \Box A$ 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 $\mathcal{C} \rightarrow \mathcal{D}$).

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

Proposition 7. *If $\mathcal{C}$ and $\mathcal{D}$ are two C.R., then $\mathcal{C} \rightarrow \mathcal{D}$, $\mathcal{C} \times \mathcal{D}$ and $\Box\mathcal{C}$ are C.R. too.*

Interpretation of types and contexts

Following the sketch of normalization proofs 'à la Tait', we define the interpretations of types.

Definition 8 (Interpretations of types).

- $[\![L_j]\!] := \{(\Delta, M) \mid \Delta \vdash M : L_j$ and M is SN in Δ $\}$,
- $[\![A \rightarrow B]\!] := [\![A]\!] \rightarrow [\![B]\!]$,
- $[\![A \times B]\!] := [\![A]\!] \times [\![B]\!]$,

– If A is not pure, $[\![\Box A]\!] := \Box[\![A]\!]$

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

Proposition 9 ($[\![L_j]\!]$ is a C.R.).
The set $[\![L_j]\!]$ is a reducibility candidate.

Proof
(CR1), (CR1'), (CR2) and (CR4) are immediate. (CR3) is easy: if M is neutral and if forall M' such that $\Delta \vdash M \hookrightarrow M' : L_j$ without η-expansion, $M' \in [\![L_j]\!]$, then we notice that actually all the reducts of M belong to $[\![L_j]\!]$ since M (of ground type L_j) cannot be η-expanded. Thus M is SN in Δ and $M \in [\![L_n]\!]$. □

In the previous definition, there is a missing case. How to define $[\![\Box A]\!]$, if A is pure? In this case we have to take into account the fact that $\Box A$ may be the type of the inductive argument of Case/It.

Let us call $\mathcal{S}$ the set of the total functions that map a pure type A to a C.R. for type A. We define the operator F from $\mathcal{S}$ to $\mathcal{S}$ by giving $F(g)\left(\overset{i=p}{\underset{i=1}{\Pi}} T_i.L\right)$ for $g \in \mathcal{S}$, where T_i are pure types and L is a ground type:

$$F(g)\left(\overset{i=p}{\underset{i=1}{\Pi}} T_i.L\right) = \bigcap_{\sigma,\tau} \left\{ \begin{array}{l} (\Delta, M) \mid \Delta \vdash M : \Box(T_1 \to \dots T_p \to L) \text{ and} \\ \forall \Delta' \text{ such that } \Delta, \Delta' \text{ is valid },\\ \forall ((\Delta,\Delta'), M_{j,k}) \in \overset{q=n_{j,k}}{\underset{q=1}{\Pi}} g\left(\overset{i=p}{\underset{i=1}{\Pi}} T_i.B_{j,k,q}\right).\tau(L_j),\\ \forall ((\Delta,\Delta'), M'_{j,k}) \in \overset{q=n_{j,k}}{\underset{q=1}{\Pi}} \tau(B_{j,k,q}).\tau(L_j),\\ ((\Delta,\Delta'), \langle\sigma\rangle\text{Case } M \text{ of } (M_{j,k})) \in \overset{z=p}{\underset{z=1}{\Pi}} \left(\overset{m=r_z}{\underset{m=1}{\Pi}} g\left(\overset{i=p}{\underset{i=1}{\Pi}} T_i.T_z^m\right).\tau(L_z)\right).\tau(L),\\ ((\Delta,\Delta'), \langle\sigma\rangle\text{It } M \;\text{ of }\; (M'_{j,k})) \in \overset{i=p}{\underset{i=1}{\Pi}} \tau(T_i).\tau(L),\\ \text{and } ((\Delta,\Delta'), \downarrow M) \in [\![T_1 \to \dots T_p \to L]\!] \end{array} \right\}$$

where σ is quantified over the total functions from ground types to types and τ is quantified over the total functions mapping each ground type L_j to a C.R.$_{\sigma(L_j)}$.

We need to prove that F is an operator from $\mathcal{S}$ to $\mathcal{S}$:

Proposition 10 (F is an operator from $\mathcal{S}$ to $\mathcal{S}$).
If g belongs to $\mathcal{S}$ then $F(g)$ belongs to $\mathcal{S}$, i.e. for every pure type A, $F(g)(A)$ is a C.R.

Proof
Consider a pure type A and a function g in $\mathcal{S}$, that is a function from pure types to C.Rs for those types. We prove that $F(g)(A)$ is a C.R. for A by showing that it is the intersection of CRs (i.e sets verifying conditions (CR1) to (CR4)) for A. (See [Lel97] for the complete proof.)

□

After proving that F is monotone, we are able to give a rigorous definition of $[\![\Box A]\!]$ when A is pure, involving the smallest fixpoint of F:

Definition 11 (Definition of $[\![\Box A]\!]$, A pure).
We call Y_F the smallest fixpoint of F. By definition, for any pure type A, $[\![\Box A]\!] := Y_F(A)$.

Of course, we have to justify this definition. First, we notice that $\mathcal{S}$ is an inf-semi lattice because the set of C.R. for a given type is an inf-semi lattice. Now, the existence of Y_F is proved by Tarski's lemma: F is a monotone operator from the inf-semi lattice $\mathcal{S}$ to the inf-semi lattice $\mathcal{S}$. Note that $[\![A]\!]$, image of A by Y_F is itself a C.R.

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 12 ($[\![A]\!]$ 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 13 (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 14 (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 2.2.

In the definition of $[\![\Delta; \Gamma]\!]_{\Psi}$, the requirement that ρ belongs to $[\![\Delta]\!]_{\Psi^n}$, 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 $[\![\Delta]\!]_{\Psi}$.

Soundness of typing Now we reach the final stage of the proof. The Strong Normalization property will be an easy corollary of the key lemma "Soundness of Typing":

Lemma 15 (Soundness of Typing).
If $\Delta \vdash M : A$ and $\rho \in [\![\Delta]\!]_{\Psi}$, then $(\Psi, \rho(M)) \in [\![A]\!]$.

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 [Lel97]).

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

3.3 Confluence and conservative extension

The *confluence* property is a corollary of the strong normalization (Theorem 16) and the local confluence results (this fact is often called "Newman's Lemma", after [New42]).

Theorem 17 (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 18 (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 the following technical lemma, which needs some notations.

Notation. Given a stack $\Delta \equiv \Gamma_1; \ldots; \Gamma_n$, two variables x and y declared respectively in Γ_i and Γ_j, we say that y is declared in the *future* of x if $j \geq i$.

A term is said to be *pure* if it is a term of the simply-typed λ-calculus (that is to say it does not contain any Case/It, any modal operators, any pairs, nor any projections).

Lemma 19 (Possible forms of a canonical term).
Given a stack Δ and its local context Γ, if $\Delta \vdash M : B$, M canonical form then:

- *a) either M contains an impure variable of Δ and all the free variables of M are in the future of an impure variable,*
- *b) or M is an impure abstraction (i.e. the type of the bound variable is impure) and all the free variables of M belong to Γ,*
- *c) or M is of the form $\uparrow N$ and N is closed,*
- *d) or M is of the form $\langle M, N \rangle$ and all the free variables of M and N belong to Γ,*
- *e) or M is pure and B is pure. Moreover all the free variables of M belong to Γ.*

Theorem 20 (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.

4 Related works

Our system has been inspired by [DPS97]. The main difference is that the underlying modal λ-calculus is easier to use and seems to be better adapted to a future extension to dependent types. Splitting the context in two parts (the intuitionistic and the modal parts) would most probably make the treatment of dependant types even more difficult: how should we represent a modal type which depends 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.

Raymond McDowell and Dale Miller have proposed [MM97] 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 give a typing system, supporting the judgments-as-types principle, but two logics: one for each level (object and meta). 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([PS98]). 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.

From our definition of valid terms in an object language $L_n = L \to \cdots \to L \to L$ implemented in the Calculus of Inductive Constructions, we derived an induction principle, that we claimed to be more natural, and more powerful, than the usual ones ([DH94]). Martin Hofmann recently formalized this induction principle in a modal meta-logic, using categorical tools [Hof99]. In this paper, he very nicely formalizes and compares, on the categorical level, several representations of terms using HOAS, and several induction principles currently used, sometimes without justifications, for fonctional terms.

5 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 [DPS97]. 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.

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 dependant types, only with a "non-dependant" rule for eliminatin for the moment [DL99,Lel98b]. A full treatment of dependant types would have given an induction principle that we did not succeed in justifying in our setting. The work by Martin Hofmann [Hof99] suggests that we should be able to go further in this direction.

Another interesting direction of research consists in replacing our recursive operators by operators for pattern-matching such as those used in the ALF [MN94] system, implementing Martin-Löf's Type Theory [NPS90]. Some hints for a concrete syntax for that extension have been given in [DPS97]. F. Pfenning and C. Schürmann are currently working on the definition of a meta-logic along these lines.

Acknowledgments Thanks are due to Martin Hofmann for his proposition for 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

[Aka93] Y. Akama. On Mints' reduction for ccc-calculus. In *Proceedings TLCA*, pages 1–12. Springer-Verlag LNCS 664, 1993.

[BdP96] Gavin Bierman and Valeria de Paiva. Intuitionistic necessity revisited. In *Technical Report CSRP-96-10*, School of Computer Science, University of Birmingham, June 1996.

[CK93] R. Di Cosmo and D. Kesner. A Confluent Reduction for the Extensional Typed λ-calculus. In *Proceedings ICALP'93*. Springer-Verlag LNCS 700, 1993.

[DH94] J. Despeyroux and A. 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), Kiev, Ukraine, July 16–21, 1994*, volume 822. Springer-Verlag LNAI, 1994.

[DL99] Joëlle Despeyroux and Pierre Leleu. Primitive recursion for higher-order abstract syntax - dependant types. Draft, submitted for publication, 1999.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Jr. Guy Steele, editor, *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg Beach, Florida, January 1996. ACM Press.

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

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. In J.R. Hindley and P. de Groote, editors, *Int. Conf. on Typed lambda calculi and applications - TLCA'97*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS 1210.

[Gha96] Neil Ghani. Eta Expansions in System F. Technical Report LIENS-96-10, LIENS-DMI, Ecole Normale Superieure, 1996.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[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.

[JG95] C.B. Jay and N. Ghani. The Virtues of Eta-Expansion. *Journal of Functional Programming*, 5(2):135–154, April 1995.

[Lel97] Pierre Leleu. A Modal Lambda Calculus with Iteration and Case Constructs. Technical Report RR-3322, INRIA, France, December 1997. http://www.inria.fr/RRRT/RR-3322.html.

[Lel98a] Pierre Leleu. Metatheoretic Results for a Modal Lambda Calculus . Technical Report RR-3361, INRIA, France, 1998. http://www.inria.fr/RRRT/RR-3361.html.

[Lel98b] Pierre Leleu. *Syntaxe abstraite d'ordre supérieur et récursion dans les théories typées.* Phd thesis, École Nationale des Ponts et Chaussées (ENPC), December 1998. In French.

[Luo94] Zhaohui Luo. *Computation and Reasoning.* Oxford University Press, 1994.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In *Proceedings of LICS'97*, pages 434–445, Warsaw, 1997.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[New42] M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Ann. Math.*, 43(2):223–243, 1942.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[PM92] Ch. Paulin-Mohring. Inductive definitions in the system coq. rules and properties. In J.F. Groote M. Bezem, editor, *Proceedings of the Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93*, Springer-Verlag LNCS 664, 1992.

[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 Sematics*, New Orleans, Louisiana, March 1995. To appear in *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris 7, 1994.