Developing (Meta)Theory of lambda-calculus in the Theory of Contexts

Electronic Notes in Theoretical Computer Science 58 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume58.html 22 pages

Developing (Meta)Theory of λ-calculus in the Theory of Contexts [1]

Marino Miculan

Dipartimento di Matematica e Informatica, Università di Udine, Italy
mailto: miculan@dimi.uniud.it

Abstract

We present a case study on the formal development of a non trivial (meta)theory in the *Theory of Contexts* using the Coq proof assistant. The methodology underlying the Theory of Contexts for reasoning on systems presented in HOAS is based on an *axiomatic* syntactic standpoint. We feel that one of the main advantages of this approach, is that it requires a very low logical overhead.

The object system we focus on is the *lazy, call-by-name λ-calculus* (λ_{cbn}), both untyped and simply typed. We will see that the formal, fully detailed development of the theory of λ_{cbn} in the Theory of Contexts introduces a small, sustainable overhead with respect to the proofs "on the paper". Moreover, this will allow for comparison with similar case studies developed in other approaches to the metatheoretical reasoning in higher-order abstract syntax.

Keywords: higher-order abstract syntax, induction, logical frameworks.

Introduction

In recent years there has been growing interest in developing systems for defining and reasoning on languages featuring α-conversion. A promising line of approach has focused on *Higher-Order Abstract Syntax* (HOAS) [13, 24]. The gyst of this approach is to delegate to type-theoretic metalanguages the burden of dealing with binders. This approach however has some drawbacks. First of all, being equated to metalanguage variables, object level variables cannot be defined inductively without introducing *exotic terms* [7, 20]. A similar difficulty arises with contexts, which are rendered as functional terms. Reasoning by induction and definition by recursion on object level terms is therefore problematic. Various approaches have been proposed to overcome these problems based on different techniques such as functor categories, permutation models of ZF, etc. [9, 10, 14, 11, 19].

[1] Work partially supported by Italian MURST project TOSCA and EC-WG TYPES.

In [15] another logical framework for reasoning on systems in HOAS is presented, based on an *axiomatic* syntactic standpoint. This system stems from the technique originally used in [16] for formally deriving in Coq [17] the metatheory of strong late bisimilarity of the π-calculus. This framework consists of a simple types theory *à la* Church extended with a set of axioms called the *Theory of Contexts*, recursion operators and induction principles. According to our experience, this framework is rather expressive. Higher Order Logic allows for the impredicative definition of many relations, possibly functional, the recursors and induction principles allow for the definition of many important functions and properties over contexts, and most notably the axioms in the Theory of Contexts allow for a smooth handling of schemata in HOAS. In fact, we feel that one of the main advantages of the axiomatic approach of the Theory of Contexts, is that it requires a very low mathematical and logical overhead.

Of course there are some tradeoffs. One of the major theoretical problems concerning any axiomatic approach is the consistency of the axioms. In fact, it is possible to prove that the Theory of Context is sound. We refer the reader to [4], where a full proof of consistency of the Theory of Contexts is given, using a model of *functor categories* along the line of [14].

From a practical point of view, however, the applicability of this approach has to be tested by means of several cases studies. These are particularly useful for discussing pragmatic issues which arise in developing real proofs in the Theory of Contexts, addressing possible solutions and suggesting directions for future developments. This is precisely the aim of this paper, where we develop a non trivial (meta)theory of the *lazy, call-by-name* λ*-calculus* (λ_{cbn}), both untyped and simply typed, in the Theory of Contexts within the Coq proof assistant. We choose λ_{cbn} as the object of this case study for several reasons. First, it is a well-known logic, so much that most works about λ-calculus skim quickly standard definitions and basic proofs, sweeping many details under the rug. We will see that the formal, fully detailed development of the theory of λ_{cbn} in the Theory of Contexts introduces a small, sustainable overhead with respect to the proofs "on the paper". Moreover, λ_{cbn} owns some features (as substitution of terms for variables, typing system,...) which are somewhat complementary to those of π-calculus, which has been the object of another large case study in the Theory of Context [16]. Finally, some variant of the λ-calculus has been always taken as the traditional benchmark/example of application of the many approaches to HOAS in the literature; see [13, 2, 7, 19, 14, 11, 12] among the others.

It turns out that Theory of Contexts is quite successful in handling the metatheory of λ_{cbn}. The encoding of the syntax, the semantics and the type system is straightforward, and still we delegate the α-conversion to the metalevel. Only the encoding of substitution is not immediate, since we need to represent it as a relation. However, this will allow us to state and prove some fundamental results (i.e., functionality of substitution) which usually are

taken for granted in informal works on the λ-calculus. Moreover, these proofs cast some light on the limits, and suggest possible solutions, of the Theory of Contexts in Coq.

The Coq code is available at `http://www.dimi.uniud.it/~miculan/HOAS/`.

Synopsis. In Section 1 we recall briefly (that is, as in most works is done) the object system λ_{cbn}. In Section 2 we give a brief presentation of the type theory CIC and its implementation Coq. The HOAS encoding of syntax and semantics of λ_{cbn}, and the formal development of its metatheory using the Theory of Contexts, is described in Section 3. Practical issues which arise in developing the proofs are discussed and, when possible, solutions are proposed. In Section 4 we consider two extensions of the object system: applicative bisimulation and observational equivalence, and a simple type system for λ_{cbn}. Comparison with related work and conclusions are in Sections 5, 6 respectively.

1 The object system λ_{cbn}

In this section we give an intentionally brief (and somewhat sloppy) definition of the lazy call-by-name λ-calculus λ_{cbn}, as it is usually given in most papers. This will allow us to enlighten, in the following sections, how the formal representation of the theory does not introduce a substantial overhead despite the high level of detail required. We assume the reader familiar with the basic notions of λ-calculus; for an introduction and a comprehensive development of the theory of the λ-calculus, see [3].

The set Λ of terms of λ_{cbn} is defined by the following grammar:

$$M, N ::= x \mid (MN) \mid \lambda x.M$$

where $x, y, z, \ldots$ range over an infinite set of variables. Terms are taken up-to α-equivalence. We denote by $M[N/x]$ the capture-avoiding substitution of N for x in M. *Contexts*, i.e. terms with holes, are denoted by $M(\cdot)$. A term is said to be a *value* if it is not an application. The notion of "free variables" (FV) is defined as usual. For X a finite set of variables, we define $\Lambda_X \triangleq \{M \in \Lambda \mid FV(M) \subseteq X\}$. By Λ^0 we denote $\Lambda_\emptyset$.

We consider two lazy operational semantics. The *reduction* (or *small-step* semantics) is the smallest relation $M \longrightarrow N$ defined by the following rules

$$\frac{}{(\lambda x.M)\ N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{(M\ N) \longrightarrow (M'\ N)}$$

We denote by $\longrightarrow^*$ the reflexive and transitive closure of $\longrightarrow$.

The *evaluation* (or *big-step* semantics) is the smallest relation $M \Downarrow N$ defined by the following rules:

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.M' \quad M'[N/x] \Downarrow V}{M\ N \Downarrow V}$$

2 The Calculus of Inductive Constructions

The Calculus of Inductive Constructions is an extension of the *Calculus of Constructions (CC)*, which can be defined as the PTS λC of Barendregt's λ-cube, with two sorts, *Prop* and *Set*. Under the *proposition-as-types, proofs-as-terms* paradigm, there is an isomorphism between propositions of intuitionistic higher-order logic and types of sort *Prop*. If A has type *Prop* then it represents a logical proposition; the fact that A is inhabited by a term M represents the fact that A holds. Each term M inhabiting A represents a *proof* of A. On the other hand, the sort *Set* is supposed to be the type of datatypes, such as naturals, lists, trees, booleans, etc. These types differ from those inhabiting *Prop* for their constructive contents.

Therefore, CC, as many similar Type Theories, can be fruitfully used as a general logic specification language, i.e. as a Logical Framework (LF) [13, 22, 23]. In an LF, we can represent faithfully and uniformly all the relevant concepts of the inferential process in a logical system (syntactic categories, terms, variables, contexts, assertions, axiom schemata, rule schemata, instantiation, tactics, etc.).

The *Calculus of Inductive Constructions (CIC)* (implemented in the Coq system [17]) extends CC with some special constants which represent the definition, introduction and elimination of inductive types. For instance, the following definition of natural numbers (written in Gallina, Coq's specification language)

```
Inductive nat : Set :=  O : nat | S : nat -> nat
```

allows to define terms by "case analysis", like the following function:

```
Definition pred := [n:nat]Cases n of  O => O  |  (S u) => u  end.
```

where `[n:nat]` is Gallina notation for abstraction $\lambda n : nat$. Using these elimination schemata, Coq automatically states and proves the induction principle for each inductively defined type. For instance, the above definition yields the Peano induction principle "for free":

```
nat_ind : (P:nat->Prop)(P O) ->
                        ((n:nat)(P n)->(P (S n))) -> (n:nat)(P n)
```

where `(n:nat)` is the notation for dependent product $\prod_{n:nat}$. This feature has been extensively used in the definition of logical connectives: we need only to specify the introduction rules, and we can prove the elimination rules from the elimination principle the system automatically provides us.

However, allowing for *any* inductive definition in CIC would yield non-normalizing terms, thus invalidating the standard proof of consistency of the system. Hence, inductive definitions are subject to the *positivity condition*, which (roughly) requires that the type we are defining does not occur in negative position in the type of any argument of any constructor. This condition ensures the soundness of the system, but it rules out also many sound inductive

definitions. For instance, the following definition of λ-terms in *higher-order abstract syntax*

```
Inductive L : Set := lam : (L->L) -> L | app : L -> L -> L.
```

is not well-formed, due to the negative occurrence of `L` in the type `L->L` of the argument of `lam`.

Another problem arising from the use of higher order abstract syntax together with inductive types is that of *exotic terms*. These are λ-terms which do not correspond to any object "on the paper", despite their types correspond to some syntactic category. Exotic terms are generated when a type has a higher-order constructor over an inductive type. A simple example is the following fragment of first-order logic:

```
Inductive i : Set := zero : i | one : i.
Inductive o : Set := ff : o | eq : i->i->o | forall : (i->o)->o.
Definition weird : o := (forall [x:i](Cases x of
                                        zero => ff
                                      | one  => (eq zero zero)
                                     end)).
```

The term `weird` does not correspond to any proposition of first order logic: there is no formula $\forall x\phi$ such that $\phi\{0/x\}$ and $\phi\{1/x\}$ are syntactically equal to "ff" and "$0 = 0$", respectively. Exotic terms are problematic in establishing the faithfulness of the formalization; usually, they have to be ruled out by means of auxiliary "validity" judgements [7, 27]. Another approach, which will be used in Section 3.1 is to have the higher order constructors to range over types which are not defined as inductive, so that there is no `Cases` to use as above.

A common implementation of CIC is Coq, an interactive proof assistant developed by the INRIA and other institutes. For a complete description, we refer to [17]. Coq is an editor for interactively searching for an inhabitant of a type, in a top-down fashion by applying tactics step-by-step, backtracking if needed, and for verifying correctness of typing judgements. A proof search starts by entering

```
Lemma ident : goal.
```

where *goal* is the type representing the proposition to prove. At this point, Coq waits for commands from the user, in order to build the proof term which inhabits *goal* (i.e., the proof). To this end, Coq offers a rich set of *tactics*, e.g., introduction and application of assumptions, application of rules and previously proved lemmata, elimination of inductive objects, inversion of (co)inductive hypotheses and so on. These tactics allow the user to proceed in his proof search much like he would do informally. At every step, the type checking algorithm ensures the soundness of the proof. When the proof term is completed, it can be saved (by the command `Qed`) for future applications.

3 Formalizing and reasoning on $\lambda_{\mathbf{cbn}}$ in CIC

3.1 Encoding the syntax of λ_{cbn}

The HOAS representation of the syntax of λ_{cbn} is the following:

```
Parameter Var : Set.
Inductive tm : Set :=  var : Var -> tm
                     |  app : tm -> tm -> tm
                     |  lam : (Var -> tm) -> tm.
Coercion var : Var >-> tm.
```

Declaring `var` as a coercion allows us to inject implicitly terms from type `Var` into `tm`, so that in the following this constructor may be omitted from terms.

Remark 3.1 We do not define `Var` as an inductive set. In fact, this is not required by the syntax of λ_{cbn}, so there is no reason to bring in unnecessary assumptions, i.e., the induction and recursion principles. Actually, these unwanted principles are not harmless, because they can be exploited for defining exotic terms; hence, taking `Var` as inductive would be simply wrong.

Remark 3.2 Notice that `lam` is a *higher-order* constructor, that is it takes a functional term as argument. In particular, terms of type `Var->tm` represent exactly the *capture-avoiding contexts* of the λ-calculus. This technique allows to inherit the α-equivalence on terms from the metalanguage, and still to have an inductive definition for terms. For instance, $\lambda x.(xx)$ and $\lambda y.(yy)$ are represented by `(lam [x:Var](app (var x) (var x)))` and `(lam [y:Var](app (var y) (var y)))`, respectively, which are the same term up-to α-conversion. At the same time we can define functions by first-order recursion or case analysis on the syntax of terms, like the following:

```
Definition isvalue := [M:tm]Cases M of
                             (var x)       => False
                           | (lam t)       => True
                           | (app t1 t2)   => False
                      end.
```

The adequacy of this encoding is a consequence of [15, Theorem 1]. For $X = \{x_1, \ldots, x_n\}$ a finite set of variables, let us define

$$\Gamma_X \triangleq \{\mathtt{x1} : \mathtt{Var}, \ldots, \mathtt{xn} : \mathtt{Var}\} \cup \{\mathtt{dij} : \tilde{}(\mathtt{xi} = \mathtt{xj}) \mid \mathtt{1} \leq \mathtt{i} < \mathtt{j} \leq \mathtt{n}\}$$
$$\mathtt{tm}_X \triangleq \{M \mid \Gamma_X \vdash M : \mathtt{tm}, M \text{ in long } \beta\eta\text{-normal form}\}.$$

Proposition 3.3 *For all X finite set of variables, there is a bijection ϵ_X between Λ_X and $\mathtt{tm}_X$. Moreover, this bijection is compositional, in the sense that if $M \in \Lambda_{X,x}$ and $N \in \Lambda_X$, then $\epsilon_X(M[N/x]) = \epsilon_{X,x}(M)[\epsilon_X(N)/\mathtt{(var\ x)}]$.*

As a corollary, a (capture-avoiding) context $M(\cdot) \in \Lambda_X$ is naturally encoded as `[z:Var]`$\epsilon_{X,z}(M(z))$, where the fresh variable z acts as a "placeholder" for the hole. In fact, a bijection like in Proposition 3.3 can be established between contexts and terms of type `Var->tm`.

3.2 The Theory of Contexts for λ_{cbn}

Often, a HOAS encoding of an object system is not sufficient for handling the *metatheory* of the system, that is to prove properties *on* the object system itself. Since we aim to prove several results about λ_{cbn}, we need to introduce the corresponding Theory of Contexts.

Given the signature of the syntax, the Theory of Contexts is composed by two parts. The first contains the definitions of "occurrence" predicates. These definitions are immediately derived from the signature of the language, following the pattern in [16, 15].

```
Inductive notin [x:Var] : tm -> Prop :=
    notin_var : (y:Var)~x=y->(notin x y)
  | notin_app : (M,N:tm)(notin x M) -> (notin x N)
                   -> (notin x (app M N))
  | notin_lam : (M:Var->tm)((y:Var)~x=y->(notin x (M y)))
                   -> (notin x (lam M)).

Inductive isin [x:Var] : tm -> Prop :=
    isin_var : (isin x x)
  | isin_app1: (M,N:tm)(isin x M) -> (isin x (app M N))
  | isin_app2: (M,N:tm)(isin x N) -> (isin x (app M N))
  | isin_lam : (M:Var->tm)((y:Var)(isin x (M y)))
                   -> (isin x (lam M)).
```

The only thing we need to know about names (variables), is that equality over `Var` is decidable. However, we do not need a full blown classical logic: it is sufficient to have a classical behaviour on the occurrence check predicates.

```
Axiom LEM_OC: (M:tm)(x:Var)(isin x M)\/(notin x M).
```

This implies the decidability of `(eq Var)`.

The second part of the Theory of Contexts consist of a set of axiom schemata, which reflect at the theory level some fundamental properties of the intuitive notion of "context" and "occurrence" of variables. Their informal meaning is the following:

Unsaturability of variables: no term can contain all variables; i.e., there exists always a variable which does not occur free in a given term;

Extensionality of contexts: two contexts are equal if they are equal on a fresh variable; that is, if $M(x) = N(x)$ and $x \notin M(\cdot), N(\cdot)$, then $M = N$.

More formally, these are the axioms of the Theory of Contexts we need:

```
Axiom unsat : (M:tm)(Ex [x:Var](notin x M)).
Axiom ext_tm : (M,N:Var->tm)(x:Var)
        (notin x (lam M)) -> (notin x (lam N)) ->
        (M x)=(N x) -> M=N.
Axiom ext_tm1 : (M,N:Var->Var->tm)(x:Var)
```

```
        (notin x (lam [z:Var](lam (M z)))) ->
        (notin x (lam [z:Var](lam (M z)))) ->
        (M x)=(N x) -> M=N.
```

The following are immediate consequences of the Theory of Contexts and the induction principles over `tm`.

```
Lemma differ : (x:Var)(Ex [y:Var]~x=y).
Lemma isin_notin_absurd : (x:Var)(M:tm)(isin x M)->(notin x M)->False.
```

Remark 3.4 In [15] also another axiom schema, called *β-expansion*, is presented. Informally, β-exp states that given a term M and a variable x, there is a context $N(\cdot)$ such that $N(x) = M$ and x does not occur in $N(\cdot)$. This has been used several times in the development of the metatheory of π-calculus [16]. On the other hand, it has not been needed in the present work on λ_{cbn}. A possible motivation is that here we are allowed for higher-order induction, while in [16] we had to recover it from induction over plain, first-order terms.

Another useful application of β-exp which we have not used in this work is the representation of variable-capturing contexts in HOAS. In general, this is difficult because bound variables are not accessible from "outside" the abstraction, not even their names. In the Theory of Contexts, a context $C[\cdot]$ capturing x can be represented as a term of type `(var->tm)->tm`, and the instantiation requires a β-expansion of the inserted term. For instance, $\lambda x.([\cdot]\, x)$ is represented as `C' = [N:(var->tm)](lam [x:Var](app (N x) (var x)))`. Consider a term M with a free variable x which has to be captured: let $M'(\cdot)$ the context obtained by β-expansion over x (i.e., such that $M'(x) = M$ and x does not appear free in M'). Then, the variable-capturing instantiation $C[M]$ is rendered as `(C' M')`.

Remark 3.5 As described in Section 2, `Coq` and similar systems provide support for inductive types. However, this does not hold for *higher-order types*: there is no induction principle over `A->B`, even if `A` and/or `B` are inductive. This is because the intended meaning of `A->B` (usually, a function space) is not an initial algebra. Thus, most proof editors give no induction principles, case analysis, inversion predicates and similar tools for reasoning on terms of type `Var->tm`, i.e., contexts. Nevertheless, it is possible to prove that types of the form `Var->...->Var->tm` do have recursion and induction principles [14,4,15]. Hence, beside the simple `Axiom`s of the Theory of Contexts above, we can safely assume higher-order induction and recursion principles as needed, like the following induction over `Var->tm`:

```
Axiom tm_ind1 : (P:(Var->tm)->Prop)
        (P var) ->
        ((y:Var)(P [_:Var](var y))) ->
        ((M,N:Var->tm)(P M)->(P N)->(P [x:Var](app (M x) (N x)))) ->
        ((M:Var->Var->tm)
```

```
          ((y:Var)(P [x:Var](M x y)))->(P [x:Var](lam (M x))))
    -> (M:Var->tm)(P M).
```

(notice how there are two base cases). In the following we may introduce also others of these principles,

3.3 Formalizing the substitution

A drawback of using using a specific type for variables (a "weak HOAS" encoding), is that we cannot delegate the substitution to the metalanguage. Instead, we need to define it by hand, as a parametric relation between contexts and terms:

```
Inductive subst [N:tm] : (Var->tm) -> tm -> Prop :=
      subst_var  : (subst N var N)
    | subst_void : (y:Var)(subst N [_:Var]y y)
    | subst_app  : (M1,M2:Var->tm)(M1',M2':tm)
                   (subst N M1 M1') -> (subst N M2 M2') ->
                   (subst N [y:Var](app (M1 y) (M2 y)) (app M1' M2'))
    | subst_lam  : (M:Var->Var->tm)(M':Var->tm)
                   ((z:Var)(subst N [y:Var](M y z) (M' z))) ->
                   (subst N [y:Var](lam (M y)) (lam M')).
```

Thus, a term M' is syntactically equal to the substitution $M(x)[N/x]$ iff `(subst N M M')` holds. More formally:

Proposition 3.6 *Let X be a finite set of variables and x a variable not in X. Let $N, M' \in \Lambda_X$ and $M \in \Lambda_{X \uplus \{x\}}$. Then:*

$$M[N/x] = M' \iff \Gamma_X \vdash \_ : (\mathit{subst}\ \epsilon_X(N)\ \mathit{[x:Var]}\epsilon_{X\uplus\{x\}}(M)\ \epsilon_X(M'))$$

Capture-avoiding substitution is naturally defined as a function, but in fact its definition is seldom given in full detail—and we made no exception in Section 1. Actually, the definition which is usually intended is not deterministic *a priori*, since it requires an arbitrary α-conversion of bound variables of the context in order to avoid capturing free variables in the substituted term. More complex languages (e.g., dynamic logic, Hoare logic [20]) may require nonstandard substitutions involving contrived notions of conversion, not simply α-conversion. Representing the substitution in CIC as a relation, in fact, gives raise to the possibility that it may be not total, that is for some N, M there is no M' such that $M' = M[N/x]$. The fact that such a definition does give a functional (i.e., deterministic and total) relation is a property which we are going to prove explicitly using the Theory of Contexts.

Determinism of substitution

The property we want to prove is the following:

```
Parameter N:tm.
```

```
Lemma subst_is_det: (M:Var->tm)(M1:tm)(subst N M M1) ->
                                  (M2:tm)(subst N M M2) -> (M1 = M2).
```

We give two proofs of this property. The first goes by induction on the derivation of `(subst N M M1)`. This gives rise to four cases:

```
  N : tm                              subgoal 2 is:
  M : var->tm                          (y)=M2
  M2 : tm                             subgoal 3 is:
  H : (subst N Var M2)                 (app M1' M2')=M0
  ============================        subgoal 4 is:
   N=M2                                (lam M')=M2
```

each of which should be dealt by inverting the hypothesis `H` (or corresponding). Usually, such an inversion would eliminate automatically all absurd cases, but this does not work when the terms which have to be discriminate are higher-order. This is indeed the case, since the second argument of `subst` has type `Var->tm`. The `Inversion H` tactic gives us four cases for the first goal, only one of which is trivially true and the other are absurd:

```
subgoal 1 is:                         subgoal 2 is:
  N : tm                                N : tm
  M : Var->tm                           M : Var->tm
  M2 : tm                               M2 : tm
  H : (subst N var M2)                  H : (subst N var M2)
  H0 : var=var                          y : Var
  H1 : N=M2                             H1 : ([_:var](y))=var
  ============================          H0 : (y)=M2
   M2=M2                                ============================
...                                      N=(y)
```

Absurd cases are (tediously) eliminated by using the Theory of Contexts, in particular the axiom of extensionality. The whole proof is 95 lines long, most of which are dealing with the elimination of absurd cases.

Determinism of substitution, again

A much shorter proof can be obtained by proving a suitable *higher-order inversion lemma* for substitution. In Coq, inversion lemmata are automatically synthesized and proved on-the-fly from recursion principles by the `Inversion` tactic, using the algorithm originally implemented by Murthy with subsequent elaboration by Cornes and Terrasse [6]. However, this algorithm fails to give the right inversion predicate when the datatype, which we have to discriminate over, is higher-order, because usual inductive type theories do not recognize a higher-order type as inductive. Nevertheless, we know that types of the form `Var->tm` do have recursion principles [14, 4, 15]. Hence, we can consistently introduce these principles (as `Axioms`) for the definition of the recursive map needed in the inversion predicate:

```
Parameter subst_inv_fun  : tm -> (Var->tm) -> tm -> Prop.
Axiom subst_inv_fun_var0 : (N,M:tm)(subst_inv_fun N var M)==(N=M).
Axiom subst_inv_fun_var1 :
```

```
         (y:Var)(B,N:tm)(subst_inv_fun N [_:Var]y B)==((var y)=B).
Axiom subst_inv_fun_app : (A1,A2:Var->tm)(B,N:tm)
         (subst_inv_fun N [x:Var](app (A1 x) (A2 x)) B) ==
              (EX B1 | (EX B2 | (app B1 B2)=B /\ (subst N A1 B1)
                                               /\ (subst N A2 B2))).
Axiom subst_inv_fun_lam : (A:Var->Var->tm)(B,N:tm)
 (subst_inv_fun N [x:Var](lam (A x)) B) ==
    (EX A1 | (lam A1)=B /\ (y:Var)(subst N [x:Var](A x y) (A1 y))).
```

Then, the higher-order inversion principle is "mechanically" claimed and proved as follows:

```
Lemma subst_inv : (A:Var->tm)(B,N:tm)(subst N A B) -> (subst_inv_fun N A B).
Intros; Inversion_clear H.
Rewrite subst_inv_fun_var0; Reflexivity.
Rewrite subst_inv_fun_var1; Reflexivity.
Rewrite subst_inv_fun_app; Exists M1'; Exists M2'; Auto.
Rewrite subst_inv_fun_lam; Exists M'; Auto.
Qed.
```

Using this inversion lemma, the proof of determinism of substitution is much easier—in fact, we "lift" at the level of context the syntactic machinery of inversion tactics that Coq provides at the level of terms:

```
Lemma subst_is_det': (M:Var->tm)(M1:tm)(subst N M M1) ->
                                (M2:tm)(subst N M M2) -> (M1 = M2).
Induction 1; Intros.
Generalize (subst_inv ? ? ? H0); Rewrite subst_inv_fun_var0; Trivial.
Generalize (subst_inv ? ? ? H0); Rewrite subst_inv_fun_var1; Trivial.
Generalize (subst_inv ? ? ? H4); Rewrite subst_inv_fun_app; Intros.
Inversion_clear H5; Inversion_clear H6; Inversion_clear H5; Inversion_clear H7.
Rewrite (H1 ? H5); Rewrite (H3 ? H8); Assumption.
Generalize (subst_inv ? ? ? H2); Rewrite subst_inv_fun_lam; Intros.
Inversion_clear H3; Inversion_clear H4; Inversion_clear H3.
Replace x with M'; Auto.
Elim (unsat (app (lam M') (lam x))).
Intros; Inversion_clear H3.
Apply ext_tm with x0; Auto.
Qed.
```

As one can see, in this proof we used also the axioms of the Theory of Contexts (`unsat` and `ext_tm`).

Totality of substitution

The proof of totality is tricky due to some peculiarities of CIC. The lemma we want to prove is

```
Lemma subst_is_total :  (M:Var->tm)(EX M' | (subst N M M')).
```

Our intent is to prove this by higher-order induction over `M`. This fails in the case of the `lam`bda abstraction, which appears as follows:

```
  N : tm
  M : Var->tm
  M0 : Var->Var->tm
  H : (y:Var)(EX M':tm | (subst N [x:Var](M0 x y) M'))
  ============================
   (EX M':tm | (subst N [x:Var](lam (M0 x)) M'))
```

The suitable term should be obtained from the hypothesis `H`. However, Coq does not allow us to eliminate a `Prop`osition (like `H`) to build a term in a Set (`M'` in `tm`). Such "eliminations of strong Σ-types" may lead to inconsistencies, and hence are ruled out by the type theory CIC [5].

The solution we adopt is to move the whole proof in the `Set` realm, and then to lift the result to `Prop`. Therefore, we introduce a `Set`-typed version of the induction principle—which, equivalently, can be seen as a recursor with dependent types:

```
Axiom tm_rec1 : (P:(Var->tm)->Set)
        (P var) ->
        ((y:Var)(P [_:Var](var y))) ->
        ((M,N:Var->tm)(P M)->(P N)->(P [x:Var](app (M x) (N x)))) ->
        ((M:Var->Var->tm)
            ((y:Var)(P [x:Var](M x y)))->(P [x:Var](lam (M x))))
          ->
        (M:Var->tm)(P M).
```

Then, we prove the totality in `Set` by higher-order dependent recursion:

```
Lemma sit: (N:tm)(M:Var->tm){M':tm | (subst N M M')}.
Intros; Pattern M; Apply tm_rec1; Intros; Clear M.
Split with N; Apply subst_var.
Exists (var y); Apply subst_void.
(Inversion_clear H; Inversion_clear H0).
Split with (app x x0); Apply subst_app; Assumption.
Exists (lam [y:Var](proj1_sig tm ? (H y))).
Apply subst_lam; Intros.
Apply proj2_sig.
Qed.
```

Notice that in the case of `lam`bda, the required term is built by eliminating (projecting) the Σ-type in the hypothesis `H` instantiated on a locally bound (and hence, fresh) variable `y`.

Then, the totality theorem is just the extraction of the logical part from the Σ-type `(sit M)`:

```
Lemma subst_is_total :  (M:Var->tm)(Ex [M':tm](subst N M M')).
Intros. Exists (proj1_sig ? ? (sit M)).
Apply proj2_sig.
Qed.
```

Extracting the substitution function

Lemma `sit` can be seen as the *specification* of the substitution function. We can make it explicit by extracting the first component of the Σ-type `(sit N M)`:

```
Lemma subst_f : tm->(Var->tm)->tm.
Intros N M; Exact (proj1_sig ? ? (sit N M)).
Qed.
```

which, sweetened by a bit of syntactic sugar, takes the familiar form `_[_]`, like in the following "verification" and congruence properties:

```
Lemma subst_f_prop : (N:tm)(M:Var->tm)(subst N M  M[N]).
Lemma subst_f_var  : (N:tm)(var[N])=N.
Lemma subst_f_void : (N:tm)(y:Var)(([_:Var]y)[N])=y.
Lemma subst_f_app  : (N:tm)(M1,M2:Var->tm)
              (([x:Var](app (M1 x) (M2 x)))[N])=(app M1[N] M2[N]).
Lemma subst_f_lam  : (N:tm)(M:Var->Var->tm)
 (([x:Var](lam (M x)))[N]) = (lam ([y:Var](([x:Var](M x y))[N]))).
```

An interesting property of substitution is *composition*: for M, N, P terms and $x \neq y$, we have that $(P[Q/y])[M/x] = (P[M/x])[Q[M/x]/y]$. The formalization of this statement requires a context with 2 holes:

```
Lemma subst_f_comp : (M:tm)(Q:Var->tm)(P:Var->Var->tm)
 (([x:Var]((P x)[(Q x)]))[M]) = ([y:Var]([x:Var](P x y))[M])[Q[M]]).
```

Beside using axioms `ext_tm` and —unsat—, the proof of this property goes by structural induction over the structure of P; thus we need to assume the corresponding induction principle on `Var->Var->tm`:

```
Axiom tm_ind2 : (P:(Var->Var->tm)->Prop)
 (P [x,y:Var]x) ->
 (P [x,y:Var]y) ->
 ((z:Var)(P [_;_:Var](var z))) ->
 ((M,N:Var->Var->tm)(P M)->(P N)->(P [x,y:Var](app (M x y) (N x y)))) ->
 ((M:Var->Var->Var->tm)
     ((z:Var)(P [x,y:Var](M x y z)))->(P [x,y:Var](lam (M x y))))
 ->
 (M:Var->Var->tm)(P M).
```

3.4 Formalizing the semantics of λ_{cbn}

The representation of both operational semantics is straightforward.

```
Inductive red : tm -> tm -> Prop :=
      red_beta: (N:tm)(M:Var->tm)
                (red (app (lam M) N) (subst_f N M))
    | red_head: (M,N,M':tm)
                (red M M') -> (red (app M N) (app M' N)).
```

```
Inductive trred : tm -> tm -> Prop :=
    | trred_ref : (M:tm)(trred M M)
    | trred_trs : (M,N:tm)(red M N)->(P:tm)(trred N P)->(trred M P).
Inductive eval : tm -> tm -> Prop :=
      eval_var : (x:Var)(eval x x)
    | eval_lam : (M:Var->tm)(eval (lam M) (lam M))
    | eval_app : (M,N,V:tm)(M':Var->tm)
                 (eval M (lam M')) -> (eval M'[N] V) ->
                 (eval (app M N) V).
```

Proposition 3.7 *Let X be a finite set of variables; for all $M, N \in \Lambda_X$, we have:*

(i) $M \longrightarrow N \iff \Gamma_X \vdash \_ : (\mathtt{red}\ \epsilon_X(M)\ \epsilon_X(N))$

(ii) $M \longrightarrow^* N \iff \Gamma_X \vdash \_ : (\mathtt{trred}\ \epsilon_X(M)\ \epsilon_X(N))$

(iii) $M \Downarrow N \iff \Gamma_X \vdash \_ : (\mathtt{eval}\ \epsilon_X(M)\ \epsilon_X(N))$

Most interesting properties of small-step and big-step semantics can be proved without using the Theory of Contexts. These properties include the progress lemma, determinism of semantics, equivalence of big-step and small-step semantics:

```
Definition closed : tm -> Prop := [M:tm](x:Var)(notin x M).
Lemma progress : (M:tm)(closed M)->(isvalue M)\/(EX N | (red M N)).
Lemma red_is_det : (M,V1:tm)(red M V1)->(V2:tm)(red M V2)->V1=V2.
Lemma eval_is_det : (M,V1:tm)(eval M V1)->(V2:tm)(eval M V2)->V1=V2.
Lemma red_eval : (M,N:tm)(red M N)->(V:tm)(eval N V)->(eval M V).
Lemma trred_eval : (M,V:tm)(trred M V)->(isvalue V)->(eval M V).
Lemma eval_trred : (M,N:tm)(eval M N) -> (trred M N).
```

These properties are proved by simple induction on the syntax of `M` and the derivation of `(red M V1)`, `(eval M V1)`, `(red M N)`, `(trred M V)`, `(eval M N)`, respectively.

4 Extending the (meta)theory

In this section we consider two extensions of the λ_{cbn}: applicative bisimulation and observational equivalence, and a simple type system. This allows us to test the *modularity* of the Theory of Contexts. It turns out that, while in the first case the Theory of Contexts previously introduced works fine, in the latter case we have to modify slightly the `unsat` axiom, in order to take into account the new behaviour of variables brought in by the type system.

4.1 Applicative Bisimulation and Observational Equivalence

In this section we extend the theory and the metatheory of λ_{cbn} by considering two well-known equivalences over λ_{cbn} [1]:

Definition 4.1 The *observational equivalence* is the relation $\approx_o$ over Λ^0 defined as follows: $M \approx_o N$ iff for all $C[\cdot] \in \Lambda^0$, if $C[M]\Downarrow$ then $C[N]\Downarrow$.

The *applicative bisimulation* is the relation $\approx_a$ over Λ^0 defined as follows: $M \approx_a N$ iff if $M \Downarrow \lambda xP$ for some P, then there exists Q such that $N \Downarrow \lambda xQ$ and for all $R \in \Lambda^0$: $P[R/x] \approx_a Q[R/x]$.

Notice that the definition of the applicative bisimulation is *coinductive*. There is also a "linearized" definition of the same relation, which we denote by $\approx'_a$: $M \approx'_a N$ iff for all n and $R_1, \ldots, R_n \in \Lambda^0$: if $(MR_1 \ldots R_n)\Downarrow$ then $(NR_1 \ldots R_n)\Downarrow$.

It is quite clear that $\approx_a = \approx'_a$. A more interesting and important result is the so called *operational extensionality*, that is $\approx_a = \approx_o$ [1]. The proof sketch given in [1] consists in the proof of the following technical lemma:

$$M \approx_a N \Rightarrow \forall n. \forall C \in \Lambda^0. C[M]\Downarrow^n \Rightarrow C[N]\Downarrow^n \tag{1}$$

where $\Downarrow^n$ is a "indexed" version of the evaluation semantics. The intended meaning of the index is the number of substitution performed in the evaluation. The rules for $\Downarrow^n$ are the following:

$$\frac{}{\lambda x.M \Downarrow^0 \lambda x.M} \qquad \frac{M \Downarrow^m \lambda x.M' \quad M'[N/x] \Downarrow^n V}{M\,N \Downarrow^{m+n+1} V}$$

Our aim is to encode these relations in CIC and to prove formally their equivalence. The formalization of $\approx_a$ and $\approx_o$ is not problematic; in particular, we can take advantage of `Coinductive` types for encoding $\approx_a$:

```
CoInductive appsim : tm -> tm -> Prop :=
      appsim_coind : (M,N:tm)
              ((M':Var->tm)(eval M (lam M')) ->
               (EX N' | (eval N (lam N')) /\
                        (L:tm)(closed L) -> (appsim M'[L] N'[L])
              -> (appsim M N).

Definition conv := [M:tm](EX V | (eval M (lam V))).
Definition obseq : tm -> tm -> Prop := [M,N:tm](C:Var->tm)
                  (closed (lam C))-> (conv C[M]) -> (conv C[N]).
```

For $\approx'_a$, on the other hand, we need to introduce the datatype of *applicative lists*, and the operation of *list application*:

```
Inductive ltm : Set := nil : ltm | larg : ltm -> tm -> ltm.
Fixpoint lapp [M:tm;L:ltm] : tm :=
        Cases L of   nil  => M
           |  (larg L' N) => (app (lapp M L') N)
        end.
Definition appsim' : tm -> tm -> Prop := [M,N:tm]
     (L:ltm)(lclosed L)-> (conv (lapp M L)) -> (conv (lapp N L)).
```

Then, two implications are proved without using the Theory of Contexts:

```
Lemma obseq_appsim'  : (M,N:tm)(obseq M N) -> (appsim' M N).
Lemma appsim'_appsim : (M,N:tm)(appsim' M N) -> (appsim M N).
```

The former is proved by induction on the applicative list in the definition of `appsim'`. The latter is proved by *coinduction*: using the tactic `Cofix` we can "assume" the conclusion and apply the coinductive rule:

```
  appsim'_appsim : (M,N:tm)(appsim' M N)->(appsim M N)
  M : tm
  N : tm
  H : (appsim' M N)
  P : Var->tm
  H0 : (eval M (lam P))
  ============================
   (EX Q:Var->tm |
      (eval N (lam Q))/\((R:tm)(closed R)->(appsim P[R] Q[R])))
```

The proof proceeds then by elimination of the hypothesis `H`. Notice that for these proofs we do not need the hypothesis that M,N are closed.

More difficult is the proof of the third inclusion, namely

```
Variable M,N: tm.
Hypothesis closedN : (closed N).
Hypothesis closedM : (closed M).
Lemma appsim_obseq : (appsim M N) -> (obseq M N).
```

For this end, we need to encode $\Downarrow^n$ and prove the property (1):

```
Inductive neval : nat -> tm -> tm -> Prop :=
      neval_lam : (M:Var->tm)(neval O (lam M) (lam M))
    | neval_app : (M,N,V:tm)(M':Var->tm)(n,m:nat)
                  (neval n M (lam M')) -> (neval m M'[N] V) ->
                  (neval (S (plus n m)) (app M N) V).
[...]
Lemma Context : (n:nat)(C:Var->tm)(closed (lam C)) ->
                      (V:tm)(neval n C[M] V) -> (conv C[N]).
```

Following [1], the proof is by induction on `n`. Induction over `Var->tm` is used for destructing the context `C` in order to obtain the two cases mentioned in [1]:

- $C[\cdot] \equiv (\lambda x P[\cdot])(Q[\cdot])\boldsymbol{R}[\cdot]$
- $C[\cdot] \equiv [\cdot](Q[\cdot])\boldsymbol{R}[\cdot]$

Beside these we obtain also other four cases which have been omitted in [1], and which are easily dealt with. The proof proceeds by several technical manipulations of applicative lists and some structural properties of indexed evaluation, like the following

```
Lemma neval_one_step : (P:Var->tm)(Q:tm)(R:ltm)(V:tm)(n:nat)
```

```
(neval (S n) (lapp (app (lam P) Q) R) V) ->
                                  (neval n (lapp P[Q] R) V).
```

In many points, the Theory of Contexts is used to prove easily equivalences between contexts.

It should be noticed that the burden of the proofs is in the manipulation of applicative lists, rather than in dealing with contexts. Many properties of lists which are usually taken for granted, here must be spelled out and proved in full detail.

4.2 Typing system

In this section we extend the theory and the metatheory of λ_{cbn} by adding *simple types*. Simple types are defined by the grammar $\tau ::= u \mid \tau \to \tau$, where u, v range over type variables. The typing judgement has the form $\Gamma \vdash M : \tau$, where Γ is the typing base, that is a finite set of pairs $x_1 : \tau_1, \ldots, x_n : \tau_n$. The usual typing rules are the following:

$$\frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} x \notin dom(\Gamma)$$

The syntax of simple types is encoded trivially:

```
Parameter tyVar : Set.
Inductive ty : Set := tvar : tyVar -> ty | arr : ty -> ty -> ty.
Coercion tvar : tyVar >-> ty.
```

The introduction of a typing system has a bearing on the structure of `Var`. In the previous section, we assumed `Var` to be simply any set with the required properties, i.e. the `unsat` axiom and, ultimately, decidability of equality. The typing system, however, requires that every free variable is given a type. This is reflected in the encoding by adding more structure to `Var`, i.e. by assuming the existence of a type assignment and by requiring that every fresh variable introduced by `unsat` must be given a type:

```
Parameter typevar : Var -> ty.
Axiom unsat_t : (M:tm)(s:ty)(EX x | (notin x M) /\ (typevar x)=s).
```

Then, the encoding of the typing system is straightforward:

```
Inductive type : tm -> ty -> Prop :=
      type_var : (x:Var)(type (var x) (typevar x))
    | type_app : (M,N:tm)(s,t:ty)(type M (arr s t)) ->
                 (type N s) -> (type (app M N) t)
    | type_lam : (M:Var->tm)(s,t:ty)
                 ((x:Var)((typevar x)=s) -> (type (M x) t))
                 -> (type (lam M) (arr s t)).
```

Notice that `unsat_t` entails `unsat`, so the results previously proved, still hold with this axiom.

The first important property we need to prove is that substitution preserves types of terms:

```
Lemma subst_preserves_types : (E:Var->tm)(N,M:tm)(subst N E M) ->
        (s,t:ty)(type (lam E) (arr s t)) -> (type N s) -> (type M t).
```

This can be proved by induction on the derivation of `(subst N E M)`, using `unsat_t` for introducing fresh names with the right types and the following lemma of *type invariance* under replacement of free variables:

```
Lemma type_invar : (M:Var->tm)(s,t:ty)
            (x:Var)((typevar x)=s) -> (type (M x) t) ->
            (y:Var)((typevar y)=s) -> (type (M y) t).
```

This lemma states a property of contexts, so it is natural to proceed "by higher-order induction on `M`". Indeed, using the higher-order induction principle `tm_ind1`, the lemma `type_invar` can be proved straightforwardly. Then, all subject reductions properties are simple consequences of `subst_preserves_types`:

```
Lemma SR_eval : (M,V:tm)(eval M V)->(s:ty)(type M s)->(type V s).
Induction 1; Clear H V M; Intros; Try Assumption.
Inversion_clear H4; Apply H3.
Apply subst_preserves_types with s:=s0 1:=H1; Auto.
Qed.

Lemma SR_red : (M,N:tm)(red M N)->(s:ty)(type M s)->(type N s).
Lemma SR_trred : (M,N:tm)(trred M N) -> (s:ty)(type M s) -> (type N s).
```

The proofs of the latter properties are similar to former's.

5 Related work

The case study presented in this paper should be compared with similar developments on the λ-calculus in HOAS. Due to the lack of space, here we can discuss briefly only some of them. Despeyroux *et al.* in [7] adopted an approach similar to ours for reasoning on the (call-by-value) λ-calculus. The main difference is that variables are represented by an *inductive* set (such as `nat`). This allows to avoid to assume the properties of the Theory of Contexts as axioms, but at the price to cope with exotic terms. These are ruled out by means of a well-formedness predicate *valid*; all arguments are then carried out on terms which are *extensionally equivalent* to some valid term. Since CIC is not extensional, this equivalence has to be embedded into *valid* explicitly. Dealing with well-formedness predicats introduces a substantial overhead with respect to informal proofs, although it is conceivable that much can be automatized by means of *ad hoc* tactics. The Coq code of [7] (covering syntax, substitution, big-steps semantics, typing system and proof of subject reduction) is 500 lines long; the same theory, developed in the Theory of Contexts, takes less than 300 lines.

On the other hand, Röckl *et al.* proved recently that the Theory of Contexts (for the π-calculus) can be derived from well-formedness predicates in Isabelle [27]. In some sense, therefore, the Theory of Contexts can be seen as a core set of basic properties capturing the essence of what a context is, without assuming unnecessary assumptions. Further case studies like the present one are needed, in order to verify the expressive power of this approach.

In [26], Pitts introduced the *Nominal Logic*, a first-order logic specifically designed for reasoning on syntax involving variable bindings. The axioms of Nominal Logic express the key properties of the FM-model of syntax introduced in [11]. The main idea is to express and deal only with properties whose validity is invariant under swapping of bindable names. It is interesting to see that many principles are common to Nominal Logic and the Theory of Contexts (e.g., unsaturability of names, "structural induction modulo α", ...). The exact connection between the two theories is still to be investigated.

As already pointed out, most nowadays proof editors, and in particular all those based on type theory, lack of induction on higher order types. The problem of combining HOAS and induction, avoiding the arising of exotic terms, has been addressed radically in [9, 8], where *modal* λ-calculi are proposed as metalanguages in place of usual type theories. In fact, this approach is best seen as a step towards a brand new generation of proof assistants, rather than a way for dealing with HOAS in nowadays proof editors, as the Theory of Contexts is intended to be.

In all these approaches, we reason on objects of the metalogic (CIC, HOL,...), in the metalogic itself. A different perspective is to add explicitly an extra logical level for reasoning over metalogics. One of these meta-metalogic is $FO\lambda^{\Delta\mathrm{N}}$ [19], a higher-order intuitionistic logic extended with definitions and higher order quantification over simply typed λ-terms. Induction on types is recovered from induction on natural numbers via appropriate notions of measure. Differently from the approach adopted in this work and in [7], in $FO\lambda^{\Delta\mathrm{N}}$ it is possible to delegate even the substitution to the metalanguage. $FO\lambda^{\Delta\mathrm{N}}$ has been successfully used to reason with typing and evaluation judgements for the call-by-name λ-calculus, among other case studies. However, $FO\lambda^{\Delta\mathrm{N}}$ is more in the streamline of logic programming: it does not support a notion of "proof object", nor the typical judgements-as-types paradigm. Actually, all the properties on λ_{cbn} proved in [19] have been proved also in the present work using the Theory of Contexts and simple (*i.e.*, structural) induction over proofs. Due to its logic programming flavour, $FO\lambda^{\Delta\mathrm{N}}$ seems more suited for a complete automatization, i.e. the implementation of a theorem prover.

A similar attitude, but with different aims, is behind Schürmann's $\mathcal{M}_2$ [25], which is a constructive first-order logic based on the Edinburgh LF. At the meta-metalevel, $\mathcal{M}_2$ offers higher-order induction and recursion for reasoning over (possibly open) objects of a LF encoding. $\mathcal{M}_2$ is aimed to a complete automatization (it is implemented in the theorem prover *Twelf*), hence it is difficult to compare with interactive approaches like the ones previously discussed.

6 Conclusions

In this paper we have presented a non trivial case study of the (meta)theory of λ_{cbn} using the Theory of Contexts in the Coq proof environment. This approach allowed for a smooth treatment of the syntax up-to α-conversion, substitution, small-step and big-step semantics and typing system of λ_{cbn}. We have formally proved several metatheoretical properties such as functionality of substitution, determinism of big-step semantics, equivalence of big-step and small-step semantics, subject reductions for both semantics, equivalence of applicative bisimulation and observational equivalence, etc.

In our opinion, the case study has been successful. The logical overhead which is required to the user is acceptable, since the encodings are straigthforward and the user can directly transpose his/her intuition and hand-made proofs about contexts in the proof editor. This case study has pointed out also some weak points of the Theory of Contexts in the Coq system, in particular in connection with higher-order inversion principles. We have shown, by means of an example, how suitable inversion principles over higher-order types can be stated and derived.

Although in this work we have focused on the Coq system, all arguments should be applicable to any other proof assistant based on inductive type theories close to CIC, such as LEGO or Plastic. On the other hand, the Theory of Contexts is inconsistent with the Axiom of Unique Choice (AC!) [14]. In particular, this means that the approach we have adopted in this paper cannot be adopted in Isabelle/HOL because the Description Axiom entailes AC!. For a comparison of formalizations of languages with variable bindings in Isabelle, see [21]. Nevertheless, it is still possible to use the Theory of Contexts within classical HOL; see [15, 4] for an example metalanguage with full classical higher-order logic, and the proof of its consistency.

Future work.

The present development of the metatheory of λ_{cbn} can be extended in many directions. A possible future work could be the generalization of the inversion algorithms presented in [6] to suitable higher-order types. From a practical point of view, this would be particularly useful.

On the theoretical side, at least two interesting issues has arisen. The first is that we have not needed the whole Theory of Contexts, since the axiom of β-expansion has not been introduced. This seems to point out that the properties of λ_{cbn} we dealt with do not rely on such kind of property. The second issue is that, due to soundness constraints imposed by CIC, in order to prove totality of substitution we introduced higher-order recursion with *dependent types*. Known models of the Theory of Contexts validate plain higher-order recursion; it is an open question if these models can be extended to dependent type theory.

References

[1] Abramsky, S. and C.-H. L. Ong, *Full abstraction in the lazy lambda calculus*, Information and Computation **105** (1993), pp. 159–267.

[2] Avron, A., F. Honsell, I. A. Mason and R. Pollack, *Using Typed Lambda Calculus to implement formal systems on a machine*, Journal of Automated Reasoning **9** (1992), pp. 309–354.

[3] Barendregt, H., "The lambda calculus: its syntax and its semantics," Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[4] Bucalo, A., M. Hofmann, F. Honsell, M. Miculan and I. Scagnetto, *Consistency of the theory of contexts* (2001), submitted.

[5] Coquand, T., *Metamathematical investigations of a calculus of constructions*, , **31**, Academic Press, 1990 pp. 91–122.

[6] Cornes, C. and D. Terrasse, *Automating inversion of inductive predicates in coq*, in: S. Berardi and M. Coppo, editors, *Proc. of TYPES'95*, number 1158 in Lecture Notes in Computer Science (1995), pp. 85–104.

[7] Despeyroux, J., A. Felty and A. Hirschowitz, *Higher-order syntax in Coq*, in: *Proc. of TLCA'95*, Lecture Notes in Computer Science **905** (1995), also appears as INRIA research report RR-2556, April 1995.

[8] Despeyroux, J. and P. Leleu, *Primitive recursion for higher-order abstract syntax with dependant types*, in: *FLoC'99 IMLA workshop*, Trento, Italy, 1999.

[9] Despeyroux, J., F. Pfenning and C. Schürmann, *Primitive recursion for higher order abstract syntax*, Technical Report CMU-CS-96-172, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (1996).

[10] Fiore, M. P., G. D. Plotkin and D. Turi, *Abstract syntax and variable binding*, in: Longo [18], pp. 193–202.

[11] Gabbay, M. J. and A. M. Pitts, *A new approach to abstract syntax with variable binding*, Formal Aspects of Computing **?** (2001), pp. ?–?, special issue in honour of Rod Burstall. To appear.

[12] Gordon, A. and T. Melham, *Five axioms of alpha-conversion*, in: *Proc. of TPHOL'96*, number 1152 in Lecture Notes in Computer Science, 1996, pp. 173–190.

[13] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the ACM **40** (1993), pp. 143–184.

[14] Hofmann, M., *Semantical analysis of higher-order abstract syntax*, in: Longo [18], pp. 204–213.

[15] Honsell, F., M. Miculan and I. Scagnetto, *An axiomatic approach to metareasoning on systems in higher-order abstract syntax*, in: *Proc. ICALP'01*, number 2076 in LNCS (2001), pp. 963–978, also available at `http://www.dimi.uniud.it/~miculan/Papers/`.

[16] Honsell, F., M. Miculan and I. Scagnetto, *π-calculus in (co)inductive type theory*, Theoretical Computer Science **253** (2001), pp. 239–285, first appeared as a talk at TYPES'98 annual workshop.

[17] INRIA, "The Coq Proof Assistant," (2000), `http://coq.inria.fr/doc/main.html` .

[18] Longo, G., editor, "Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science," The Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press, Trento, Italy, 1999.

[19] McDowell, R. and D. Miller, *Reasoning with higher-order abstract syntax in a logical framework*, ACM Transactions on Computational Logic (2001), to appear.

[20] Miculan, M., "Encoding Logical Theories of Programs," Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Italy (1997).

[21] Momigliano, A., S. Ambler and R. Crole, *A comparison of formalizations of the meta-theory of a language with variable bindings in Isabelle*, Technical Report 2001/07, University of Leicester (2001).

[22] Paulin-Mohring, C., *Inductive definitions in the system Coq; rules and properties*, in: M. Bezem and J. F. Groote, editors, *Proc. of Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science **664** (1993), pp. 328–345.

[23] Pfenning, F., *The practice of Logical Frameworks*, in: *Proc. CAAP'96*, number 1059 in Lecture Notes in Computer Science (1996), pp. 119–134.

[24] Pfenning, F. and C. Elliott, *Higher-order abstract syntax*, in: *Proc. of ACM SIGPLAN '88 Symposium on Language Design and Implementation*, The Association for Computing Machinery, Atlanta, Georgia, 1988, pp. 199–208.

[25] Pfenning, F. and C. Schürmann, *System description: Twelf — A meta-logical framework for deductive systems*, in: H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, number 1632 in LNAI (1999), pp. 202–206.
URL `citeseer.nj.nec.com/pfenning99system.html`

[26] Pitts, A., *A first-order theory of names and binding*, in: S. Ambler, R. Crole and A. Momigliano, editors, *Proc. MERLIN'01*, Technical Report **2001/26** (2001), pp. 1–13.

[27] Röckl, C., D. Hirschkoff and S. Berghofer, *Higher-order abstract syntax with induction in Isabelle/HOL: Formalising the π-calculus and mechanizing the theory of contexts*, in: F. Honsell and M. Miculan, editors, *Proc. FOSSACS 2001*, number 2030 in LNCS (2001), pp. 359–373.