The Theory of Contexts for First Order and Higher Order Abstract Syntax [1]

Furio Honsell and Marino Miculan and Ivan Scagnetto

Dipartimento di Matematica e Informatica, Università di Udine
Via delle Scienze 206, 33100 Udine, Italy. honsell,miculan,scagnett@dimi.uniud.it

Abstract

We present two case studies in formal reasoning about untyped λ-calculus in Coq, using both *first-order* and *higher-order abstract syntax*. In the first case, we prove the equivalence of three definitions of α-equivalence; in the second, we focus on properties of substitution. In both cases, we deal with *contexts*, which are rendered by means of higher-order terms (functions) in the metalanguage. These are successfully handled by using the *Theory of Contexts*.

Introduction

In this paper we present two case studies stemming from two well-known encoding paradigms of languages with binders in Logical Frameworks: *first-order* and *higher-order abstract syntax* (FOAS and HOAS, respectively). In the former case, we can choose between de Bruijn indexes or explicit names, but in both cases the treatment of binders is problematic: using de Bruijn indexes is a daunting task and difficult to understand (e.g., 600 technical lemmata for two binders in [9]). On the other hand, the use of explicit names charges the user with the burden of encoding the mechanism of α-equivalence. As we will see, from the point of view of computer aided proof development, this represents a non-trivial task, since α-equivalence is often taken for granted without an explicit axiomatization. Usually only a short definition is given in natural language, followed by some examples [1]. Actually, α-equivalence can be defined rigorously in more than one way; for instance, besides the "conventional" one [1], there is the variant used by McKinna-Pollack [15], and Gabbay-Pitts' alternative, based on the notion of *variable transposition* [6,7]. It is therefore natural to show formally that these three definitions are really equivalent.

[1] Research supported by the MURST Project TOSCA.

Following the higher-order approach, instead, binders are represented by means of *higher-order* constructors. Therefore, α-equivalence is automatically given by the metalanguage of the logical framework. In a type theory without native support for inductive types, like the Edinburgh LF, the natural choice for representing the "λ" binder is a *full HOAS* constructor of type $(\Lambda \to \Lambda) \to \Lambda$; this would allow us to delegate also substitution to the metalanguage [8]. However, if we want to take advantage of the inductive features of a metalanguage like CIC, we cannot resort to a full HOAS approach, because constructor types like $(\Lambda \to \Lambda) \to \Lambda$ violate the positivity condition of inductive types. Hence, we are forced to introduce a separate type for variables with the consequence that capture-avoiding substitution of terms for variables is no more delegated to the metalevel (*weak* HOAS). Moreover, the latter cannot be defined naïvely as a recursive function, but only as a relation. This raises questions about its properties, such as functionality (determinism and totality), composition, etc.

Recently, in order to deal with (meta)theory of nominal calculi, the authors have proposed a set of axioms called the *Theory of Contexts* [11], stemming from the technique originally used in [12] for formally deriving in Coq [13] the metatheory of strong late bisimilarity of the π-calculus. These axioms 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 $M = N$.

β-expansion: 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 $N[\cdot]$.

In this paper, we apply the Theory of Contexts for dealing formally with the issues above. More precisely, as far as the FOAS encoding is concerned, we formally prove that the three alternative definitions of α-equivalence are indeed equivalent; as far as the HOAS enconding is concerned, we prove the functionality of the substitution relation. We will see that in both cases the Theory of Contexts allows for a smooth treatment of the basic notion of *renaming*, when this is rendered through the functional application of the underlying metalanguage.

In conclusion, the Theory of Contexts turns out to be a powerful tool for metareasoning on nominal calculi not only if these are encoded in HOAS, but even in the case of a plain, first-order approach with explicit names.

Synopsis. In Section 1 we briefly recall the untyped λ-calculus. Then, in Section 2 we give a first-order encoding of the object language, we describe and encode the three notions of α-equivalence we take into account and we formally prove their equivalence by means of the Theory of Contexts. In Section 3 we present a higher-order encoding of untyped λ-calculus and of capture-avoiding substitution, with a formal development of the related metatheory. Final conclusions are in Section 4. The Appendix is devoted to a brief introduction to the Calculus of Inductive Constructions and to the `Coq` proof assistant.

1 The object language Λ

The set Λ of untyped λ-terms 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 $\mathcal{V}$. 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 $M[\cdot]$. The notion of "free variables" (FV) and "variables", both free and bound, (V) are 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 $\Lambda_\emptyset$. For an introduction and a comprehensive development of the theory of the λ-calculus, see [1].

2 First-Order Abstract Syntax approach

As a first case study, we will address the problem of encoding three notions of α-equivalence: the "conventional" one given in common textbooks on λ-calculi (see, e.g., [1]), a variant used in a previous formal study of the metatheory of λ-calculus [15], and an alternative formulation proposed in [6]. We are not interested here in taking a HOAS-based approach in encoding the syntax of the object language. Otherwise, α-equivalence would be provided for free by the metalanguage of the logical framework and would not be accessible at the object level.

Due to lack of space, we cannot present all the results of the complete development; we refer the interested reader to [21].

2.1 Encoding of syntax

The first-order representation of the syntax of Λ is the following:

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

It should be noticed that `Var` is not an inductive set. The only terms which can inhabit `Var` are variables of the metalanguage, which represent directly the variables of the object language. In fact, `Var` is not required to be inductive by the definition of the syntax of Λ, 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. Moreover, they are inconsistent with the properties of the Theory of Contexts we are going to use (see [12] for the details).

For $X = \{x_1, \ldots, x_n\}$, throughout the paper we will denote by Γ_X the typing environment given by $\{\mathtt{x1} : \mathtt{Var}, \ldots, \mathtt{xn} : \mathtt{Var}\} \cup \{\mathtt{dij} : \tilde{}(\mathtt{xi} = \mathtt{xj}) \mid \mathtt{1} \leq \mathtt{i} < \mathtt{j} \leq \mathtt{n}\}$. Moreover, in the remaining of this section Λ_X will denote the set $\{M \mid M \in \Lambda, V(M) \subseteq X\}$ (differently from Section 1). Finally $\mathtt{tm}_X$ will represent the canonical forms `M` (i.e. $\beta\eta$-head normal forms) of type `tm` such that $\Gamma_X \vdash_{\Sigma_\Lambda} \mathtt{M} : \mathtt{tm}$. The adequacy of the given encoding is stated by the following proposition, which is a conseguence of [11] Theorem 1 and can be proved using a standard inductive argument:

Proposition 2.1 *For each $X \subset \mathcal{V}$ finite, there is a bijection ε_X (with inverse δ_X) between Λ_X and $\mathtt{tm}_X$. Moreover, this bijection is compositional, i.e., if $M \in \Lambda_{X,x}$ and $N \in \Lambda_X$, then $\varepsilon_X(M[N/x]) = \varepsilon_{X,x}(M)[\varepsilon_X(N)/(\mathtt{var\ x})]$.*

2.2 The Theory of Contexts for Λ in FOAS

As the type `Var` is concerned we assume that the equivalence between names is decidable:

```
Axiom dec: (x,y:Var)x=y \/ ~x=y.
```

In order to introduce the remaining axioms of the Theory of Contexts, we need to define the *non occurrence* predicate:

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

The intuitive meaning of `(notin x M)` is that the variable `x` does not occur (neither free nor bound) into `M`. The definition is completely driven by the signature, following the general pattern of [12, 11].

The second axiom about the type `Var` is the unsaturation property:

```
Axiom unsat: (M:tm)(Ex [x:Var](notin x M)).
```

As a technical remark, we notice that it is possible to derive `unsat` from a more "general" unsaturation property which is independent from the peculiar syntax of the object language (`var_list` is the type of (finite) list of variables and `notin_list` is the predicate allowing to express the non occurrence of a variable in a list):

```
Axiom unsat_list: (l:var_list)(Ex [x:Var](notin_list x l)).
```

The lack of higher-order constructors in `tm` allows us to derive the axiom of β-expansion (for plain terms) by means of a structural induction on `M`:

```
Lemma EXP: (M:tm)(x:Var)
           (Ex [N:Var->tm](notin_context x N) /\ M=(N x)).
```

On the other hand, the extensionality and monotonicity axioms still have to be postulated:

```
Axiom ext: (F,G:Var->tm)(x:Var)(notin_context x F) ->
           (notin_context x G) -> (F x)=(G x) -> F=G.
Axiom notin_mono: (M:Var->tm)(x,y:Var)
                  (notin x (M y)) -> (notin_context x M).
```

where `notin_context` is an abbreviation for

```
[x:Var][F:Var->tm]((y:Var)~x=y -> (notin x (F y))).
```

Notice that monotonicity axiom has not been cited in the Introduction. Actually, if we assume unsaturability also at the level of contexts, this property could be derived by induction over the terms using the other axioms of the Theory of Contexts.

2.3 Encoding of α-equivalence (I)

We take as a starting point the following definition taken from [1]:

Definition 2.2 (i) A *change of bound variables* in M is the replacement of a part $\lambda x.N$ of M by $\lambda y.(N[x := y])$ where y does not occur (at all) in N. (Because y is fresh there is no danger in the substitution $N[x := y]$).

(ii) M is α-congruent with N, notation $M \equiv_\alpha N$, if N results from M by a series of changes of bound variables.

It is clear that a fresh renaming mechanism lies at the very heart of the notion of α-equivalence. Hence, we first introduce the following inductive predicate, implementing a simple minded renaming (i.e., it does not check if the new variable is fresh):

```
Inductive change_var [x,y:Var]: tm -> tm -> Prop :=
  change_var_var1: (change_var x y (var x) (var y))
| change_var_var2: (z:Var)~x=z ->
                   (change_var x y (var z) (var z))
| change_var_app:  (R,S,R',S':tm)(change_var x y R R') ->
                   (change_var x y S S') ->
                   (change_var x y (app R S) (app R' S'))
| change_var_lam1: (M,M':tm)(change_var x y M M') ->
                   (change_var x y (lam x M) (lam y M'))
| change_var_lam2: (M,M':tm)(z:var)~x=z -> (change_var x y M M')
                   -> (change_var x y (lam z M) (lam z M')).
```

Intuitively, `(change_var x y M N)` holds if and only if the term `N` is the result of replacing each occurrence (free or bound) of `x` with `y` into `M`. More formally, the following result states the adequacy of `change_var` w.r.t. the predicate whose inference rules are depicted in Figure 1.

Proposition 2.3 (Adequacy of change_var) *Let $X \subset \mathcal{V}$ finite, $x, y \in \mathcal{V}$*

$$\frac{-}{x[x:=y]=y}\ (\text{CV\_VAR1}) \qquad \frac{M[x:=y]=M'}{(\lambda x.M)[x:=y]=\lambda y.M'}\ (\text{CV\_LAM1})$$

$$\frac{x\neq z}{z[x:=y]=z}\ (\text{CV\_VAR2}) \qquad \frac{M[x:=y]=M' \quad x\neq z}{(\lambda z.M)[x:=y]=\lambda z.M'}\ (\text{CV\_LAM2})$$

$$\frac{M[x:=y]=M' \quad N[x:=y]=N'}{(MN)[x:=y]=M'N'}\ (\text{CV\_APP})$$

Fig. 1. Changing a variable in a λ-term.

Soundness: *if there exists* t *such that* $\Gamma_{X\cup\{x,y\}} \vdash_{\Sigma_\Lambda}$ t : (change_var x y M N)*, then we have* $\delta_X(\mathtt{M})[x:=y]=\delta_X(\mathtt{N})$*.*

Completeness: *let* $M,N\in\Lambda_X$ *such that* $M[x:=y]=N$*; then there is a canonical form* t *such that* $\Gamma_{X\cup\{x,y\}} \vdash_{\Sigma_\Lambda}$ t : (change_var x y $\varepsilon_X(M)$ $\varepsilon_X(N)$)*.*

Proof. Easily proved by means of Proposition 2.1 and by induction on the structure of the normal forms (Soundness), and induction on the structure of the derivation of the variable-changing judgment (Completeness). □

However, changing a variable "blindly" (i.e., without checking if the new one is fresh) by means of change_var could yield the problem of *capturing* occurrences of variables which were free before performing the replacement. For example, let us consider the term (lam x (app x y))$=\varepsilon_{x,y}(\lambda x.xy)$, then we can derive (change_var y x (lam x (app x y)) (lam x (app x x))) which corresponds to the following equation "on paper":

$$(\lambda x.xy)[y:=x]=\lambda x.xx$$

The free occurrence of y into $\lambda x.xy$ has been *captured* once replaced by x, while the λ-terms $\lambda x.xy$ and $\lambda x.xx$ are not α-equivalent. Hence, we must be sure that, when replacing a variable in a λ-term, the new one is fresh. The encoding of freshness is carried out by means of the notin predicate we have introduced in Section 2.2:

```
Inductive alphaBar: tm -> tm -> Prop:=
  alphaBar_var: (x:Var)(alphaBar (var x) (var x))
| alphaBar_app: (M,M',N,N':tm)(alphaBar M M') ->
             (alphaBar N N') -> (alphaBar (app M N) (app M' N'))
| alphaBar_lam1: (x:Var)(M,N:tm)(alphaBar M N) ->
                 (alphaBar (lam x M) (lam x N))
| alphaBar_lam2: (x,y:Var)(M,N:tm)(notin y M) ->
                 (change_var x y M N) ->
                 (alphaBar (lam x M) (lam y N))
| alphaBar_trans: (M,N,R:tm)(alphaBar M R) -> (alphaBar R N) ->
                  (alphaBar M N).
```

The first three constructors are congruence rules, the fourth one is the *change of bound variables* rule, while the last one is transitivity (recall that two terms can differ by one or more changes of bound variables).

Proposition 2.4 *(Adequacy of* `alphaBar`*)*

(i) *(Soundness) Let $X \subset \mathcal{V}$ finite, if* `t` *is a canonical form such that* $\Gamma_X \vdash_{\Sigma_\Lambda}$ `t` : (`alphaBar M N`)*, then we have* δ_X(`M`) $\equiv_\alpha$ δ_X(`N`).

(ii) *(Completeness) Let $X \subset \mathcal{V}$ finite, $M, N \in \Lambda_X$, then if $M \equiv_\alpha N$ there is a canonical form* `t` *such that* $\Gamma_X \vdash_{\Sigma_\Lambda}$ `t` : (`alphaBar` $\varepsilon_X(M)$ $\varepsilon_X(N)$).

Proof. Also this result can be proved straightforwardly using the same technique specified in Proposition 2.3. □

Thanks to the previous adequacy theorem, `alphaBar` can be regarded as a specification which the subsequent definitions we are going to investigate must fulfill in order to faithfully represent the notion of α-equivalence.

2.4 Encoding of α-equivalence (II)

In [15] an alternative encoding of α-equivalence is proposed; translating it in terms of our representation of the λ-calculus, we obtain the following definition:

```
Inductive alpha: tm -> tm -> Prop:=
  alphaMKP_var: (x:Var)(alphaMKP (var x) (var x))
| alphaMKP_app: (M,M',N,N':tm)(alphaMKP M M') -> (alphaMKP N N')
             -> (alphaMKP (app M N) (app M' N'))
| alphaMKP_lam: (x,y,z:Var)(M,M',N,N':tm)
             (notin z M) -> (notin z N) ->
             (change_var x z M M') -> (change_var y z N N') ->
             (alphaMKP M' N') -> (alphaMKP (lam x M) (lam y N)).
```

The first two constructors (`alphaMKP_var` and `alphaMKP_app`) are congruence rules, while the `alphaMKP_lam` constructor states that, in order to establish the equivalence of $\lambda x.M$ and $\lambda y.N$, we must pick a fresh name z (not occurring at all in M and N) and prove that $M[x := z]$ is α-equivalent to $N[y := z]$.

We formally proved the equivalence of `alphaBAR` and `alphaMKP`:

```
Lemma ALPHABAR_ALPHAMKP: (A,B:tm)(alphaBar A B)->(alphaMKP A B).
Lemma ALPHAMKP_ALPHABAR: (A,B:tm)(alphaMKP A B)->(alphaBar A B).
```

Both lemmata are proved by structural induction on the derivation of the premise. The second proof is trivial, while the first requires, as a preliminary result, to prove the transitivity of `ALPHAMKP`. In order to derive it we followed the approach pioneered in [15].

2.5 Encoding of α-equivalence (III)

In [6], the α-equivalence relation is proved to be equivalent to the binary relation $\sim$ defined by the rules depicted in Figure 2 (where $(a\ b)\cdot(-)$ stands for the operation of *variable-transposition*). As observed in [6], variable-transposition is an operation more basic than other notions of renaming (e.g. textual and

$$\frac{x \in \mathcal{V}}{x \sim x}\ \text{(GP-}\alpha\text{-VAR)} \qquad \frac{M_1 \sim M_1' \qquad M_2 \sim M_2'}{M_1 M_2 \sim M_1' M_2'}\ \text{(GP-}\alpha\text{-APP)}$$

$$\frac{(z\ x)\cdot M \sim (z\ y)\cdot M'}{\lambda x.M \sim \lambda y.M'}(z \text{ does not occur in } M, M') \quad \text{(GP-}\alpha\text{-LAM)}$$

Fig. 2. Gabbay-Pitts alternative definition of α-equivalence.

capture-avoiding substitution). Indeed, $(y\ x)\cdot M$ means that all the occurrences of x are replaced by occurrences of y and vice versa (without worrying about eventual captures).

Such a mechanism can be expressed by means of HOAS in a very natural way; indeed, if x and y are are two distinct variables occurring into M and $\varepsilon_X(M) =$ `M`, then we can derive[2] that there is a context `M':Var->Var->tm` such that `x` and `y` do not occur into `M'` and `M=(M' x y)` holds. Then the operation $(y\ x)\cdot M$ can be simply denoted by `(M' y x)`. Moreover, if y does not occur in M, we have that `M=(M'' x)` where both `x` and `y` do not occur into `M''`. Whence $(y\ x)\cdot M$ can be denoted by `(M'' y)`, without resorting to binary contexts.

Thus, the encoding of the Gabbay-Pitts formulation of α-equivalence is given by the following inductive predicate:

```
Inductive alphaGP: tm -> tm -> Prop:=
    alphaGP_var: (x:Var)(alphaGP (var x) (var x))
  | alphaGP_app: (M,M',N,N':tm)(alphaGP M M') -> (alphaGP N N')
            -> (alphaGP (app M N) (app M' N'))
  | alphaGP_lam: (M,N:Var->tm)(x,x',y:Var)(notin_context x M) ->
               (notin_context x' N) -> (notin_context y M) ->
               (notin_context y N) -> (alphaGP (M y) (N y)) ->
               (alphaGP (lam x (M x)) (lam x' (N x'))).
```

As we can see, the only differences w.r.t. the definition of `alphaMKP` are in the rule involving the `lam` constructor.

Indeed, `alphaMKP` and `alphaGP` are formally equivalent:

```
Lemma ALPHAMKP_ALPHAGP: (A,B:tm)(alphaMKP A B) -> (alphaGP A B).
Lemma ALPHAGP_ALPHAMKP: (A,B:tm)(alphaGP A B) -> (alphaMKP A B).
```

Both proofs are carried out by induction on the derivation of the premise. The only interesting cases are related to the introduction rules for the `lam` constructor: a key property in order to conclude is the following (proved by means of a structural induction on `M` and of the expansion and extensionality properties):

```
Lemma CHANGE_VAR_RW: (M,N:tm)(x,y:Var)(change_var x y M N) ->
        (M':Var->tm)(notin_context x M') -> M=(M' x)->N=(M' y).
```

[2] The derivation makes use of lemma `EXP` (presented in Section 2.2) and of the axioms of β-expansion for unary contexts and monotonicity.

In the proof of ALPHAGP_ALPHAMKP, an additional *fresh renaming* result has to be proved:

```
Lemma ALPHAMKP_RW: (A,B:tm)(alphaMKP A B) ->
                   (x,z:Var)(notin z A) -> (notin z B) ->
                   (A':tm)(change_var x z A A') ->
                   (B':tm)(change_var x z B B') -> (alphaMKP A' B').
```

The reason is that (notin_context y M) and (notin_context y N) do not necessarily imply (notin y (M x)) and (notin y (N x')) (while the vice versa is true by the monotonicity axiom). Hence, we cannot use the name y provided by the induction hypothesis, but we must pick a completely fresh variable in order to make the conclusion.

3 High-Order Abstract Syntax approach

In this section, we adopt a well-known higher-order representation of λ-calculus, using a separate type for variables (weak HOAS). It follows that we cannot define substitution as a recursive function, but only as a relation. We will prove that this relation is indeed functional.

Due to lack of space, we cannot present all the results of the complete development about this encoding; we refer the interested reader to [17].

3.1 Formalizing the syntax

The HOAS representation of the syntax of Λ 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.

As in the FOAS encoding, we do not define Var as an inductive set in order to be able to prove the adequacy of the representation and to avoid inconsistencies with the Theory of Contexts.

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.

The adequacy of this encoding, already proved in [17], is a consequence of [11, Theorem 1]. For $X = \{x_1, \ldots, x_n\}$ a finite set of variables, recall that

$\texttt{tm}_X \triangleq \{M \mid \Gamma_X \vdash M : \texttt{tm}, M \text{ in long } \beta\eta\text{-normal form}\}$.

Proposition 3.1 *For all X finite set of variables, there is a bijection ε_X between Λ_X and $\texttt{tm}_X$. Moreover, this bijection is compositional, in the sense that if $M \in \Lambda_{X,x}$ and $N \in \Lambda_X$, then $\varepsilon_X(M[N/x]) = \varepsilon_{X,x}(M)[\varepsilon_X(N)/\texttt{(var x)}]$.*

As a corollary, a (capture-avoiding) context $M[\cdot] \in \Lambda_X$ is naturally encoded as `[z:Var]`$\varepsilon_{X,z}(M(z))$, where the fresh variable z acts as a "placeholder" for the hole. In fact, a bijection like in Proposition 3.1 can be established between contexts and terms of type `Var->tm`.

3.2 Formalizing substitution

As for α-equivalence, capture-avoiding substitution is not always defined in full detail. Typically it is intended to be a (parametric) function $\cdot[N/x] : \Lambda \to \Lambda$ recursively defined on the syntax of terms, for instance as follows.

$$
\begin{aligned}
&x[N/x] = N\\
&y[N/x] = y \qquad (x \neq y)\\
&(M_1\ M_2)[N/x] = (M_1[N/x]\ M_2[N/x])\\
&(\lambda y.M)[N/x] = \lambda y.(M[N/x]) \qquad (x \neq y)
\end{aligned}
$$

It is important to notice that, in order to have a total function, this definition has to be taken up-to α-equivalence. This means that in the case of λ-abstraction, when y is exactly x, there is a "silent" (i.e., hidden) conversion of $(\lambda y.M)$ into $(\lambda z.M[z/y])$, where z can be chosen arbitrarily fresh. Thus, the definition of capture-avoiding substitution 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 [16]) may require nonstandard substitutions involving contrived notions of conversion, not simply α-conversion.

Capture-avoiding substitution could be entirely delegated to the metalanguage of a Logical Framework, if we used a *full HOAS* encoding. In such approach, there would be no specific type `Var` for variables, and the λ constructor would be rendered simply as `lam :  (tm -> tm) -> tm`. Therefore, a context would be represented as a map of type `tm -> tm`, and thus substitution becomes the application of the context to the substituting term. However, full HOAS encodings are not compatible with inductive definitions of `Coq`, because they do not satisfy the *positivity condition*, which (roughly) requires that the type we are defining (`tm`) does not occur in negative position in the type of any argument of any constructor. For this reason, we have resorted to a *weak* HOAS encoding, as the one above. A drawback of this solution, however, is that we cannot delegate the substitution to the metalanguage anymore. Instead, we need to define it by hand, as a parametric relation between contexts and terms in a logic-programming style (as in [5]):

```
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, the (proof-irrelevant) adequacy of `subst` is as follows:

Proposition 3.2 *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:*

$$M[N/x] = M' \iff \Gamma_X \vdash \_ : (\texttt{subst}\ \varepsilon_X(N)\ \texttt{[x:Var]}\varepsilon_{X\uplus\{x\}}(M)\ \varepsilon_X(M'))$$

Representing the substitution in CIC as a relation 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.

3.3 The Theory of Contexts for Λ in HOAS

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 [12, 11].

```
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 to the decidability of `(eq Var)`.

Then we can assume 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.
```

Coq and similar systems do not provide induction 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 [10,2,11,5]. Hence, beside the simple `Axioms` of the Theory of Contexts above, we can safely assume higher-order induction and recursion principles as needed (provided that `Var` is constructorless), like the following induction over `Var->tm` (notice that there are two base cases):

```
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 that we do not assume *β-expansion.* 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 [12]. On the other hand, it has not been needed in the present work on Λ. A possible motivation is that here we allowed for higher-order induction, while in [12] we had to recover it from induction over plain, first-order terms.

3.4 (Meta)Theory of Substitution

3.4.1 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 discriminated 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 deal with the elimination of absurd cases.

3.4.2 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 [4]. 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 [10, 2, 11]. 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.
[...]
Qed.
```

Using this inversion lemma, the proof of determinism of substitution is much easier (12 lines). In fact, we "lift" at the level of context the syntactic machinery of inversion tactics that Coq provides at the level of terms.

3.4.3 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 `lambda` 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 `Proposition` (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 [3].

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).
```

The soundness of such a principle can be established by porting to dependent types the model construction in [10, 2].

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.
[...]
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.
```

3.4.4 Extracting the substitution function

Lemma `sit` can be seen as the *specification* of the substitution function. We can derive it 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 with a bit of syntactic sugar, takes the familiar form `_[_]`, like in the following "verification" and congruence properties:

```
Lemma subst_f_verif: (N,V:tm)(M:Var->tm)(subst N M V) -> M[N]=V.
Lemma subst_f_var  : (N:tm)(var[N])=N.
Intro; Apply subst_f_verif; Apply subst_var.
Qed.
Lemma subst_f_void : (N:tm)(y:Var)(([_:Var]z)[N])=z.
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]))).
```

Of course, the application of the function `subst_f` cannot be actually reduced, since the recursor `tm_rec1` is an axiom and hence has no computational content. However, it is possible to "realize" it by means of an external ML program, which goes "under the hood" of Coq by discriminating between variables and abstractions. In such a case, `subst_f` would be the full-fledged capture-avoiding substitution one has in mind.

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).
```

4 Conclusions

In this paper we have briefly presented two case studies of the Theory of Contexts for dealing with properties of the syntax of λ-calculus. We have applied the Theory of Contexts to both a first-order and a (weak) higher-order encodings. In the first case, we have proved that three alternative presentations of the α-equivalence are equivalent; in the latter, we have proved some properties (such as functionality) of substitution, which has to be represented as a relation. Due to lack of space, we have not described the complete developments; we refer the interested reader to [21, 17].

It turns out that a HOAS-based approach towards substitution of variables for variables is very useful even when there are no binders in the object language (or for some reason they are not encoded by means of the binder of the metalanguage). Indeed, `alphaGP`, as an inductive definition, appears to be more clean and elegant than `alphaMKP`; moreover, the former does not depend on an auxiliary relation (`alphaMKP`, instead, depends on `change_var`).

Hence, following this approach, we have to deal with higher-order terms representing contexts, even if the syntax is represented in a plain, first-order approach. In order to handle smoothly these contexts, we have used the Theory of Contexts. This theory turned out to be particularly suited, because it embeds at the logical level those natural notions that contexts enjoy.

A The Calculus of Inductive Constructions

The *Calculus of Inductive Constructions (CIC)* 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) [8, 18, 19]. In an LF, following the "judgment-as-types" paradigm, 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 (implemented in the Coq system [13]) 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 *(full)* *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" judgments [5, 20]. Another approach, which we have used in this paper, is to have the higher order constructors to range over types which are not 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 [13]. `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 judgments. 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.

Decidability of $=$	$\forall x, y. x = y \vee x \neq y$
Decidability of $\notin$	$\forall x, M. x \in M \vee x \notin M$
Unsaturation	$\forall M. \exists x. x \notin fv(M)$
Monotonicity of $\notin$	$\forall x, y, M. x \in M[y] \Rightarrow x \notin M[\cdot]$
β**-expansion**	$\forall M, x. \exists N[\cdot]. x \notin fv(N[\cdot]) \wedge M = N[x]$
Extensionality	$\forall M[\cdot], N[\cdot], x. x \notin fv(M[\cdot]) \cup fv(N[\cdot]) \Rightarrow$ $M[x] = N[X] \Rightarrow M[\cdot] = N[\cdot]$

Fig. B.1. The Theory of Contexts.

B The Theory of Contexts

For the sake of completeness in Figure B.1 we list the properties of the Theory of Contexts in full generality, i.e., without sticking to a particular logical framework and/or encoding. Then we resume the instantiations we used in the formal development outlined throughout the paper.

B.1 Axioms for the first case study (α-equivalence)

```
Axiom dec: (x,y:Var)x=y \/ ~x=y.
Axiom unsat: (M:tm)(Ex [x:Var](notin x M)).
Axiom ext: (F,G:Var->tm)(x:Var)(notin_context x F) ->
           (notin_context x G) -> (F x)=(G x) -> F=G.
Axiom notin_mono: (M:Var->tm)(x,y:Var)
                  (notin x (M y)) -> (notin_context x M).
```

B.2 Axioms for the second case study (higher-order substitution)

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

References

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

[2] A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto. Consistency of the theory of contexts. Submitted, 2001.

[3] T. Coquand. Metamathematical investigations of a calculus of constructions. In *Logic and Computer Science*, volume 31, pages 91–122. Academic Press, 1990.

[4] C. Cornes and D. Terrasse. Automating inversion of inductive predicates in coq. In *Proc. of TYPES'95*, LNCS 1158, pages 85–104. Springer-Verlag, 1996.

[5] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order syntax in Coq. In *Proc. of TLCA'95*, LNCS 905, 1995. Springer-Verlag.

[6] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In Longo [14], pages 214–224.

[7] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, ?:?–?, 2001. To appear.

[8] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, Jan. 1993.

[9] D. Hirschkoff. Bisimulation proofs for the π-calculus in the Calculus of Constructions. In *Proc. TPHOL'97*, LNCS 1275. Springer-Verlag, 1997.

[10] M. Hofmann. Semantical analysis of higher-order abstract syntax. In Longo [14], pages 204–213.

[11] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, LNCS 2076, pages 963–978. Springer, 2001.

[12] F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[13] INRIA. *The Coq Proof Assistant.* `http://coq.inria.fr/doc/main.html` .

[14] G. Longo, editor. *Proc. 14th LICS*, Trento, Italy, July 1999.

[15] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3-4), Nov. 1999.

[16] M. Miculan. *Encoding Logical Theories of Programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, Mar. 1997.

[17] M. Miculan. Developing (meta)theory of λ-calculus in the theory of contexts. In S. Ambler, R. Crole, and A. Momigliano, ed., *Proc. MERLIN'01*, TR 2001/26, Dept. of Math. and Comp. Sci., Univ. of Leicester, pages 65–81. June 2001.

[18] C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In *Proc. TLCA*, LNCS 664, pages 328–345. Springer-Verlag, 1993.

[19] F. Pfenning. The practice of Logical Frameworks. In *Proc. CAAP'96*, LNCS 1059, pages 119–134. Springer-Verlag, April 1996.

[20] C. Röckl, 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*, LNCS 2030, pages 359–373. Springer-Verlag, 2001.

[21] I. Scagnetto. *Reasoning on Names In Higher-Order Abstract Syntax.* PhD thesis, Dipartimento di Matematica e Informatica, Università di Udine, 2002. In preparation.