Reasoning on an imperative object-based calculus in Higher Order Abstract Syntax

Alberto Ciaffaglione, Luigi Liquori, Marino Miculan

Reasoning on an Imperative Object-based Calculus in Higher Order Abstract Syntax *

Alberto Ciaffaglione‡
ciaffagl@dimi.uniud.it

Luigi Liquori†
Luigi.Liquori@inria.fr

Marino Miculan‡
miculan@dimi.uniud.it

ABSTRACT

We illustrate the benefits of using *Natural Deduction* in combination with *weak Higher-Order Abstract Syntax* for formalizing an object-based calculus with objects, cloning, method-update, types with subtyping, and side-effects, in inductive type theories such as the *Calculus of Inductive Constructions*. This setting suggests a clean and compact formalization of the syntax and semantics of the calculus, with an efficient management of method closures. Using our formalization and the *Theory of Contexts*, we can prove formally the Subject Reduction Theorem in the proof assistant Coq, with a relatively small overhead.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal definitions and Theory—*syntax, semantics*; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: [logics of programs]

General Terms

Languages, Theory, Verification.

Keywords

Interactive theorem proving, Logical foundations of programming, Program and system verification, Object-based calculi with side-effects, Logical Frameworks.

1. INTRODUCTION

Object-based languages, like Self and Obliq, are a kind of object-oriented languages where there is no notion of "class", differently from class-based languages, like Java, C++ and C#. Object-based languages are simpler, provide more primitive and flexible mechanisms, and can be used as intermediate languages in the implementation of class-based ones. Despite this, object-based languages are less used in practice than the latter; for this reason, while in recent years much effort has been put in the formalization of class-based languages, relatively little or no formal work has been carried out on object-based ones. This is a serious gap, because most of the foundational calculi introduced for the mathematical analysis of the object-oriented paradigm are object-based [2, 13]; in fact, object-based calculi represent for the object-oriented paradigm what the λ-calculus and its variants are for the functional paradigm.

The main aim of the present work is to fill this gap. More precisely, we focus on Abadi and Cardelli's impς [2], a particularly representative object-based calculus featuring objects, methods, cloning, dynamic lookup, method-update, types, subtypes, and, last but not least, imperative features. Both static and dynamic aspects of impς will be formalized in the *Calculus of Inductive Constructions* type theory (CIC), implemented in the interactive proof assistant Coq, which has been already successfully used in the formalization of class-based languages.

Due to its nature, impς is quite complex, both at the syntactic and the semantic level: impς features all the idiosyncrasies of functional languages with imperative features; moreover, the store model underlying the language allows for loops, thus making the typing system quite awkward. Consequently, key metatheoretic properties such as the *Subject Reduction* are much harder to state and prove for impς than for usual functional languages, already "on paper".

Therefore, a plain, naïve first-order encoding of impς would add further confusing details, leading to a clumsy and unmanageable formalization; hence, a key aspect of this work is that the overhead introduced by the formalization should be as low as possible. This can be achieved by taking most advantage of the metatheoretic features offered by the type theory-based Logical Frameworks (LFs). In particular, the direct support for *Natural Deduction* and *Higher-Order Abstract Syntax*, common to most LFs, can be used proficiently for defining alternative, and sparer, formalizations. For a significant fragment of impς (*i.e.*, without method-update), we can even drop the so-called *store-types* and all related machinery, by taking full advantage of *coinductive judgments*. However, in this paper we are interested in presenting the difficulties and solutions we have developed for the full impς calculus, rather than defining radically new semantics for ob-

*Research supported by Italian MIUR project COFIN 2001013518 CoMeta.

†INRIA-LORIA, Nancy, France.

‡Department of Mathematics and Computer Science, University of Udine, Italy.

ject calculi; therefore, we will not discuss coinductive types and related proof systems. We refer the reader to [6].

Natural Deduction leads us to consider a different formulation of static and dynamic semantics of impς, namely in the style of the *Natural Deduction Semantics* (NDS) [4, 22]. The key point in using NDS is that all stack-like structures (*e.g.*, environments, typing bases) are distributed in the hypotheses of the proof derivations. Therefore, these structures do not appear explicitly anymore in the formal judgments and proofs we have to deal with, which become fairly simpler. Of course, the semantics in NDS we focus on is equivalent to the original *Natural Semantics* of [2].

On the other hand, the impς calculus has *binders*, which we have to deal efficiently with both in the syntax and in the semantics (due to the presence of closures). To this end, the *weak Higher-Order Abstract Syntax* (weak HOAS) approach has emerged as one of the most suited in inductive type theories like CIC [10, 17, 18]. Following such an approach, a separate, open type for variables is introduced, and binders are represented as second-order term constructors, taking functions over variables as arguments. In this way, it is possible to delegate to the metalanguage all the burden of α-conversion, still retaining the benefits of inductive definitions. The disadvantage of weak HOAS is that it does not allow to inherit *substitution* of terms for variables from the metalanguage, as it is possible, instead, using full HOAS in non inductive type theories [15, 26]. Luckily, this is not a problem in the case of impς, since its semantics is defined using *closures* instead of substitutions. Therefore, the solution of using weak HOAS fits perfectly the needs for encoding impς; as a byproduct, the NDS and HOAS style yield a different, more efficient handling of closures.

However, the weak HOAS reveals its main drawback when we come to reasoning *over* our formalization of impς, *e.g.* for proving key metaproperties such as the Subject Reduction. Here we have to prove several properties concerning *variable renaming*, often by *induction* over second-order terms, and by *case analysis* over the name of the variables themselves. All this is problematic because CIC, and similar type theories, are not expressive enough [18].

In recent years, various approaches have been proposed to overcome these kinds of problems. One approach tends to propose *new*, more expressive metalanguages, especially designed for reasoning on names and variables, and based on different techniques such as modal types, functor categories, permutation models of set theory, *etc.* [11, 12, 14, 16]. Another line of research tends to cope with these problems *within* the current systems and logical frameworks [18, 25, 27]. A comparison of the different approaches is out of the scope of this paper.

Since we carry out a weak HOAS encoding in CIC, we have decided to adopt the *Theory of Contexts* (ToC) [17], a quite general "plug-and-play" methodology for reasoning on object systems in weak HOAS. The gist of this approach is to extend directly the existing framework (CIC, in this case) with a small set of *axioms* capturing some basic and natural properties of names and term contexts. The main advantage of such a technique is that it requires a very low mathematical and logical overhead: it can be easily used in existing proof environments without the need of any redesign of the system, and it allows for carrying out proofs with arguments similar to those "on paper". It turns out that also in the present case, the ToC reveals to be well-suited: we actually succeed in proving the Subject Reduction for impς with a small, sustainable overhead.

Our effort is useful also from the point of view of Logical Frameworks. Their theoretical development and implementation will benefit from complex case studies like the present one, where we test the applicability of advanced encoding and proof methodologies. An interesting remark is that we have to slightly modify the "unsaturation", one of the axioms of the ToC, in order to take into account the different kinds of informations kept in the proof derivation environment. This is similar to previous applications of the ToC to typed languages [23], and it is needed in order to respect a form of "regularity" of well-formed contexts. Thus, we argue that such a "regularity" of proof contexts should allow to generalize further the ToC, subsuming the several variants used in the case studies so far.

Synopsis. The paper is structured as follows. Section 2 gives a brief account of the impς-calculus, whose formalization in weak HOAS is presented in Section 3. Development of metatheoretic results, using the ToC, is discussed in Section 4. Final discussions and directions for future work are in Section 5. The Coq code is available at [7].

2. THE IMPς CALCULUS

Syntax. Abadi and Cardelli's impς-calculus [2] is an imperative calculus of objects forming the kernel of the programming language Obliq [5]. The syntax is the following:

$Term : a, b$	$::=$	x	variable
		$[l_i = \varsigma(x_i)b_i]^{i\in I}$	object
		$clone(a)$	cloning
		$a.l$	method invocation
		$a.l \leftarrow \varsigma(x)b$	method update
		$let\ x = a\ in\ b$	local declaration

Notice that ς and *let* bind x in b, so usual conventions about α-conversion apply. The reader is referred to [2, Ch.10,11] for an explanation of the intuitive meaning of these constructs. We present the semantics of impς using a Natural Deduction (NDS) style, because this leads to a simpler formalization in LFs, as will be explained and carried out in the next section.

Dynamic Semantics. First we need to introduce some implementation-level entities, such as store-locations, results, reduction contexts, closures and stores:

$Loc :$	ι	$\in$	Nat		store loc.
$Res :$	v	$::=$	$[l_i = \iota_i]^{i\in I}$		result
$RedCtx : \Gamma$		$::=$	$\emptyset \mid \Gamma, x \mapsto v$	$(x \notin \mathsf{Dom}(\Gamma))$	red. ctx.
$Store :$	s	$::=$	$\iota_i \mapsto \langle \varsigma(x_i)b_i, \Gamma_i \rangle^{i\in I}$		store

A *method-closure* $\langle \varsigma(x)b, \Gamma \rangle$ is a method together with a set of declarations for its free variables, the *(reduction) context*. In reduction contexts, variables are uniquely associated to *results* of object declarations. Each result is a (possibly empty) list of pairs: method-names together with store-locations, where the corresponding method-closure is stored. Hence a *store* is a function mapping locations to closures: in the following, the notation $\iota_i \mapsto c_i\ ^{i\in I}$ denotes the store that

maps the locations ι_i to the closures c_i, for $i \in I$; $s, \iota \mapsto c$ extends s with c at location ι (fresh), and $s.\iota_j \leftarrow c$ denotes the result of replacing the content of the location ι_j of s with c. Unless explicitly remarked, all the l_i, ι_i are distinct.

The big-step operational semantics is expressed by a reduction judgment relating a store s, a term a, a result v and another store s' (possibly different from s), in a given reduction context, *i.e.*:

$$\Gamma \vdash s \cdot a \rightsquigarrow v \cdot s'$$

The intended meaning is that, evaluating the term a from the store s in the reduction context Γ, yields the result v, in an updated store s'. In this semantics, variables never need to be replaced (*i.e.* substituted) by terms; instead, their result is book-kept in the reduction context. The rules for the reduction judgment are in Figure 1. Slightly different from the original presentation [2], ours is in Natural Deduction style (despite that rules are written in "horizontal", sequent-like format for saving space): in fact, in all the rules, the reduction context in each premise is a superset of the reduction context in the conclusion. This alternative presentation is much easier to implement and reason about in proof assistants based on type theory, as we will see in the next sections. Clearly, our presentation is equivalent to the original one (see [8, 9]).

Static Semantics. The impς-calculus has a first-order type system with subtyping. The only type constructor is that for object-types, *i.e.* $TType: \ A, B ::= [l_i : A_i]^{i \in I}$, so the only ground type is $[\,]$.

In order to define the typing judgment we need to introduce the *typing contexts* (or bases), which are unique assignments of types to variables:

$$TypCtx: \ \Delta ::= \emptyset \mid \Delta, x{:}A \quad (x \notin \mathsf{Dom}(\Delta))$$

The type system is given by three judgments: well-formedness of object-types $\Delta \vdash A$, subtyping $\Delta \vdash A <: B$, and term typing $\Delta \vdash a : A$. The rules for these judgments are collected in Figure 2; notice that also this system is in Natural Deduction style. Subtyping induces *subsumption:* an object of a given type also belongs to any super-type of that type and can subsume objects in the super-type, because these have a more limited protocol. Correspondingly, the rule $(Sub{-}Obj)$ allows a longer object-type to be a subtype of a shorter one, when the shared components have *exactly* the same types. Thus, object-types are *invariant* (*i.e.* neither covariant nor contravariant) in their component types.

The typing of results is delicate, because results contain pointers to the store, and stores may contain loops; thus, it is not possible to determine the type of a result by examining its substructures recursively. In order to cope with this problem, a whole store is typed *a priori* by means of *store-types*. This is possible because type-sound computations do not store results of different types in the same location. A store-type associates a *method-type* to each store location. Method-types have the form $[l_i : B_i]^{i \in I} \Rightarrow B_j$, where $[l_i : B_i]^{i \in I}$ is the type of "`self`" and B_j, such that $j \in I$, is the type of the j-th method-body:

M	$::=$	$[l_i : B_i]^{i \in I} \Rightarrow B_j$	method-type $(j \in I)$
Σ	$::=$	$\iota_i \mapsto {M_i}^{i \in I}$	store-type
$\Sigma_1(\iota)$	$\triangleq$	$[l_i : B_i\,]^{i \in I}$	if $\Sigma(\iota) = [l_i : B_i]^{i \in I} \Rightarrow B_j$
$\Sigma_2(\iota)$	$\triangleq$	B_j	if $\Sigma(\iota) = [l_i : B_i]^{i \in I} \Rightarrow B_j$

The type assignment system for results is given by five judgments: well-formedness of method-types $M \models \diamond$ and store-types $\Sigma \models \diamond$, result typing $\Sigma \models v : A$ and store typing $\Sigma \models s$, and context compatibility $\Sigma \models \Gamma : \Delta$. The intended meaning of the main (result typing) judgment "$\Sigma \models v : A$" is that the result v is given the type A, using the types assigned to locations by Σ. On the other hand, $\Sigma \models s$ ensures that the content of every store location of s can be given the type assigned by Σ to the same location. The rules for these judgments are collected in Figure 3.

Subject Reduction. The Type Soundness property for impς establishes that every well-typed and not diverging term never yields the "message_not_found" runtime error. This is an immediate consequence of the Subject Reduction theorem [2].

Definition 1 (Store-type extension). *We say* $\Sigma' \geq \Sigma$ *(Σ' is an extension of Σ) if and only if* $\mathsf{Dom}(\Sigma) \subseteq \mathsf{Dom}(\Sigma')$, *and for all* $\iota \in \mathsf{Dom}(\Sigma)$*:* $\Sigma'(\iota) = \Sigma(\iota)$.

Theorem 1 (Subject Reduction). *If* $\Delta \vdash a : A$, *and* $\Gamma \vdash s \cdot a \rightsquigarrow v \cdot s'$, *and* $\Sigma \models s$, *and* $\mathsf{Dom}(s) = \mathsf{Dom}(\Sigma)$, *and* $\Sigma \models \Gamma : \Delta$, *then there exist a type* A', *and a store-type* Σ', *such that* $\Sigma' \geq \Sigma$, *and* $\Sigma' \models s'$, *and* $\mathsf{Dom}(s') = \mathsf{Dom}(\Sigma')$, *and* $\Sigma' \models v : A'$, *and* $A' <: A$.

3. FORMALIZATION OF IMPς IN CIC

In this section, we present and discuss the formalization of impς in our LF based on type theory. For definiteness, we work in the *Calculus of Inductive Constructions* (CIC), and specifically in its Coq implementation, although the methodology we follow can be applied in any similar LF.

In encoding impς, we have to address efficiently several aspects of its syntax and semantics. A clean and compact encoding is a prerequisite for simplifying the development of complex metatheoretical results, such as the Subject Reduction. In general, a naïve encoding would lead to a clumsy and unmanageable set of definitions, which would add further difficulties to the formal proofs, which are already difficult on paper. Therefore, we need encoding methodologies that take most advantage of the features provided by modern type theories, so that most (if not all) details implicitly taken for granted working with paper and pencil can be automatically inherited in the formal developments.

Among other issues, the treatment of binders, closures and stores is of particular interest. We focus on these aspects at the syntactic and semantic level, in turn.

3.1 Syntax

Following the weak higher-order abstract syntax paradigm, we reduce all binders to the sole λ-abstraction. Therefore, the encoding of the syntax of terms and types of impς is the following:

```
Parameter  Var : Set. Definition Lab := nat.
Inductive Term : Set := var : Var -> Term
            | obj  : Object -> Term
            | call : Term -> Lab -> Term
            | clone: Term -> Term
            | over : Term -> Lab -> (Var->Term) -> Term
            | let  : Term -> (Var->Term) -> Term
with Object : Set := obj_nil : Object
            | obj_cons : Lab -> (Var->Term)
                              -> Object -> Object.
Coercion var : Var >-> Term.
Inductive TType:Set:=mktype:(list (Lab*TType)) -> TType.
```

$$\frac{x \mapsto v \in \Gamma \quad \forall\, \iota \in v.\ \iota \in \mathsf{Dom}(s)}{\Gamma \vdash s \cdot x \rightsquigarrow v \cdot s} \quad (Red{-}Var) \qquad \frac{\Gamma \vdash s \cdot a \rightsquigarrow v' \cdot s' \quad \Gamma, x \mapsto v' \vdash s' \cdot b \rightsquigarrow v'' \cdot s''}{\Gamma \vdash s \cdot let\ x = a\ in\ b \rightsquigarrow v'' \cdot s''} \quad (Red{-}Let)$$

$$\frac{\forall\, (x \mapsto v) \in \Gamma, \forall\, \iota' \in v.\ \iota' \in \mathsf{Dom}(s)}{\Gamma \vdash s \cdot [l_i = \varsigma(x_i)b_i]^{i \in I} \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot (s, \iota_i \mapsto \langle \varsigma(x_i)b_i, \Gamma \rangle^{i \in I})} \quad (Red{-}Obj)$$

$$\frac{\Gamma \vdash s \cdot a \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot s' \quad \iota_i \in \mathsf{Dom}(s') \quad \forall i \in I}{\Gamma \vdash s \cdot clone(a) \rightsquigarrow [l_i = \iota'_i]^{i \in I} \cdot (s', \iota'_i \mapsto s'(\iota_i)^{i \in I})} \quad (Red{-}Clone)$$

$$\frac{\Gamma \vdash s \cdot a \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot s' \quad s'(\iota_j) = \langle \varsigma(x_j)b_j, \Gamma' \rangle \quad \Gamma, \Gamma', x_j \mapsto [l_i = \iota_i]^{i \in I} \vdash s' \cdot b_j \rightsquigarrow v \cdot s'' \quad j \in I}{\Gamma \vdash s \cdot a.l_j \rightsquigarrow v \cdot s''} \quad (Red{-}Sel)$$

$$\frac{\Gamma \vdash s \cdot a \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot s' \quad \iota_j \in \mathsf{Dom}(s') \quad j \in I}{\Gamma \vdash s \cdot a.l_j \leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot (s'.\iota_j \leftarrow \langle \varsigma(x)b, \Gamma \rangle)} \quad (Red{-}Upd)$$

Figure 1: Operational semantics for impς.

Notice that we use a separate type `Var` for variables: the only terms which can inhabit `Var` are the variables of the metalanguage. Thus α-equivalence on terms is immediately inherited from the metalanguage, still keeping induction and recursion principles. If `Var` were inductive, we could define *"exotic terms"* using the `Case` construct of Coq [10]. Exotic terms are Coq terms not corresponding to any expression of impς, which therefore should be ruled out by extra "well-formedness" judgments, thus complicating the whole encoding. Notice also that the constructor `var` is declared as a coercion, thus it may omitted in the following. Weak HOAS has well-known encoding methodologies [15, 28]; therefore, the adequacy of the encoding *w.r.t.* the NDS presentation of Section 2 follows from standard arguments.

An alternative definition of objects could use directly the polymorphic lists of the Coq library, instead of introducing a separate type `Object`, as follows:

```
| obj : (list (Lab * (Var->Term))) -> Term
```

However, in our experience, this choice would not allow for defining, by recursion on the structure of terms, some fundamental functions which are essential for the rest of the formalization (such as, for example, the non-occurrence of variables "$\notin$"). Using polymorphic lists, the specification of these functions would be actually "unguarded", while they are feasible adopting the above definition.

3.2 Dynamic Semantics

As pointed out in the previous sections, a proof system in natural deduction like that in Figure 1 is easily encoded in a LF based on type theory. This can be carried out by taking advantage of hypothetical-general judgments *à la* Martin-Löf: the content of Γ, *i.e.* the bindings between variables and results, is actually represented through assumptions in the proof context, namely by means of hypothetical premises local to sub-reductions, which are discharged in the conclusion, so adhering to the natural deduction style. Thus, the evaluation judgment $\Gamma \vdash s \cdot a \rightsquigarrow v \cdot s'$ can be represented by an inductive predicate:

```
Inductive eval : Store -> Term -> Store -> Res -> Prop :=
```

but, before giving its specification, we need to address some details about locations and closures.

Locations. Locations, as method names (*i.e.*, labels), can be faithfully represented by natural numbers; this allows to define results and reduction contexts:

```
Definition Loc := nat.
Definition Res : Set := (list (Lab*Loc)).
Parameter stack : Var -> Res.
```

Informally, reduction contexts are (partial) functions, mapping (declared) variables to results; therefore, they can be represented as the graph of a function `stack`, which associates a result to each declared variable. This map is never effectively defined, but only described by a finite set of assumptions of the form "`(stack x)=v`" (where "`=`" is Leibniz's equality). Each of such declarations corresponds to an assumption "$x \mapsto v$" of the context Γ in Figure 1; these assumptions are used in evaluating variables, and discharged when needed, as in the rule for `let`:

```
e_var : (s:Store) (x:Var) (v:Res) (stack x) = (v) ->
        (eval s x s v)
e_let : (s,s',t:Store) (a:Term) (b:Var->Term) (v,w:Res)
        (eval s a s' v) ->
        ((x:Var) (stack x) = (v)->(eval s' (b x) t w)) ->
        (eval s (let a b) t w)
```

In the rule `e_let`, the "hole" of `b` is filled with a fresh (*i.e.*, locally quantified) variable associated to `v`. This rule allows also to enlighten why the weak HOAS approach is completely well-suited *w.r.t.* impς: the only substitution we need is that of variables for variables, which is completely delegated to the metalanguage.

Closures. Some remarks about the rules dealing with closures are in order. Formally, closures are pairs $\langle \varsigma(x)b, \Gamma \rangle$, where $\varsigma(x)b$ is a method and Γ is the reduction context. Therefore, we have to face two problems in implementing closures: how to represent these sets of declarations Γ, and how to gather and put back them from/to the proof context.

Since all the variables declared in the local context of a closure are "morally" bound in the body of the method, we have that the names of these "local" variables are pointless: so, reduction contexts are equivalent to local declarations of variables. Therefore, following the weak HOAS approach, we implement closures as a concrete second-order datatype:

$$\frac{\Delta \vdash A_i \quad \forall i \in I}{\Delta \vdash [l_i : A_i]^{i\in I}} \quad (Type{-}Obj) \qquad \frac{\Delta \vdash A}{\Delta \vdash A <: A} \quad (Sub{-}Refl)$$

$$\frac{\Delta \vdash A <: B \quad \Delta \vdash B <: C}{\Delta \vdash A <: C} \quad (Sub{-}Trans) \qquad \frac{\Delta \vdash A_i \quad \forall i \in I \cup J}{\Delta \vdash [l_i : A_i]^{i\in I\cup J} <: [l_i : A_i]^{i\in I}} \quad (Sub{-}Obj)$$

$$\frac{\Delta \vdash a : A \quad \Delta \vdash A <: B}{\Delta \vdash a : B} \quad (Val{-}Sub) \qquad \frac{x{:}A \in \Delta \quad \Delta \vdash A}{\Delta \vdash x{:}A} \quad (Val{-}Var)$$

$$\frac{\Delta, x_i{:}[l_i : A_i]^{i\in I} \vdash b_i : A_i \quad \forall i \in I}{\Delta \vdash [l_i = \varsigma(x_i)b_i]^{i\in I} : [l_i : A_i]^{i\in I}} \quad (Val{-}Obj) \qquad \frac{\Delta \vdash a : [l_i : A_i]^{i\in I} \quad j \in I}{\Delta \vdash a.l_j : A_j} \quad (Val{-}Sel)$$

$$\frac{\Delta \vdash a : [l_i : A_i]^{i\in I}}{\Delta \vdash clone(a) : [l_i : A_i]^{i\in I}} \quad (Val{-}Clone) \qquad \frac{\Delta \vdash a : A \quad \Delta, x{:}A \vdash b : B}{\Delta \vdash let\ x = a\ in\ b : B} \quad (Val{-}Let)$$

$$\frac{\Delta \vdash a : [l_i : A_i]^{i\in I} \quad \Delta, x{:}[l_i : A_i]^{i\in I} \vdash b : A_j \quad j \in I}{\Delta \vdash a.l_j \leftarrow \varsigma(x)b : [l_i : A_i]^{i\in I}} \quad (Val{-}Upd)$$

Figure 2: Type assignment system for terms of impς.

$$\frac{j \in I}{[l_i : B_i]^{i\in I} \Rightarrow B_j \models \diamond} \quad (Meth{-}Type) \qquad \frac{M_i \models \diamond \quad \forall i \in I}{\iota_i \mapsto M_i^{i\in I} \models \diamond} \quad (Store{-}Type)$$

$$\frac{\Sigma \models \diamond \quad \Sigma_1(\iota_i) = [l_i : \Sigma_2(\iota_i)]^{i\in I} \quad \forall i \in I}{\Sigma \models [l_i = \iota_i]^{i\in I} : [l_i : \Sigma_2(\iota_i)]^{i\in I}} \quad (Res) \qquad \frac{\Sigma \models \Gamma_j : \Delta_j \quad \forall j \in I \qquad \Delta_i, x_i{:}\Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\iota_i)}{\Sigma \models \iota_i \mapsto \langle \varsigma(x_i)b_i, \Gamma_i \rangle^{i\in I}} \quad (Store{-}Typing)$$

$$\frac{\Sigma \models \diamond}{\Sigma \models \emptyset : \emptyset} \quad (\Gamma{-}\emptyset{-}Typ) \qquad \frac{x \notin \mathsf{Dom}(\Gamma \cup \Delta) \quad \Sigma \models \Gamma : \Delta \quad \Sigma \models v : A}{\Sigma \models \Gamma, x \mapsto v : \Delta, x{:}A} \quad (\Gamma{-}Var{-}Typ)$$

Figure 3: Type assignment system for results of impς.

```
Inductive Body : Set  := ground : Term -> Body
                | bind : Res -> (Var->Body) -> Body.
Definition Closure    : Set := Var -> Body.
```

For instance, the encoding of the closure

$$\langle \varsigma(x)b, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \rangle$$

is the following term of type `Closure`:

```
[x:Var] (bind v1
      [x1:Var] (... (bind vn
             [xn:Var] (ground b)) ...))
```

where the outermost abstraction represents the "`self`" local variable x. Since stores are finite maps from locations to closures, we can define stores straightforwardly as:

```
Definition Store : Set := (list Closure).
```

Some functions are needed for manipulating the above structures (*e.g.*, for merging lists of pairs into single lists, for generating new results from objects and results, *etc.*), but their definition is not problematic; see [7] for the code.

Evaluation of closures. Let us see how closures are "opened" in the implementation of the rule for method invocation ($Red{-}Sel$). Essentially, we have to add to the current proof environment all the bindings recorded in the closure; then, the evaluation of the closure-body may take place in the extended environment. The evaluation of closure-bodies can be pre-formalized by an auxiliary judgment $\Gamma \vdash s \cdot \bar{b} \rightsquigarrow_b v \cdot s'$ whose rules are the following:

$$\frac{\Gamma \vdash s \cdot a \rightsquigarrow v \cdot s'}{\Gamma \vdash s \cdot ground(a) \rightsquigarrow_b s' \cdot v} \quad (e_b{\_}ground)$$

$$\frac{\Gamma, y \mapsto v \vdash s \cdot \bar{b} \rightsquigarrow_b v' \cdot s'}{\Gamma \vdash s \cdot bind(v, \lambda y.\bar{b}) \rightsquigarrow_b v' \cdot s'} \quad (e_b{\_}bind)$$

Correspondingly, rule ($Red{-}Sel$) can be stated better as:

$$\frac{\Gamma \vdash s \cdot a \rightsquigarrow [l_i = \iota_i]^{i\in I} \cdot s' \quad s'(\iota_j) = \lambda x_j.\bar{b}_j \quad \Gamma, \Gamma', x_j \mapsto [l_i = \iota_i]^{i\in I} \vdash s' \cdot \bar{b}_j \rightsquigarrow_b v \cdot s'' \quad j \in I}{\Gamma \vdash s \cdot a.l_j \rightsquigarrow v \cdot s''} \quad (Red{-}Sel')$$

Thus, $\rightsquigarrow$ and $\rightsquigarrow_b$ are two mutually recursive predicates:

```
Mutual Inductive
          eval : Store -> Term -> Store -> Res -> Prop :=
with eval_body : Store -> Body -> Store -> Res -> Prop :=
```

The rules for `eval_body` are immediate; let us see the encoding of the above ($Red{-}Sel'$):

```
e_call :
   (s,s',t:Store) (a:Term) (v,w:Res) (c:Closure) (l:Lab)
   (eval s a s' v) -> (In l (proj_lab_res v)) ->
   (store_nth (loc_in_res v l s') s') = (c) ->
   ((x:Var) (stack x) = (v)->(eval_body s' (c x) t w)) ->
   (eval s (call a l) t w)
```

The auxiliary functions `store_nth` and `loc_in_res` implement the dereferencing of locations in stores, and the lookup of locations in results, respectively. The closure so obtained is `c`, which is evaluated by `eval_body` after that a local variable `x`, denoting "`self`", is associated to (the implementation of) the receiver, *i.e.* host, object itself.

Construction of closures. The construction of closures is more delicate. We cannot form pairs out of the proof context "Γ" and methods: closures have to be "calculated" by wrapping method-bodies with all the results associated to their free variables, which are in the proof context. This closure construction is carried out by the auxiliary judgment $wrap \subseteq Term \times Body$. The intended meaning of $\Gamma \vdash wrap(b, \bar{b})$ is that "$\bar{b}$ is a closure-body obtained by binding all free variables in the term b to their respective results, which are in Γ". In order to keep track of free variables in terms, we introduce a judgment $closed \subseteq Term$, whose formal meaning is $\Gamma \vdash closed(a) \iff \forall x \in \mathsf{FV}(a) : closed(x) \in \Gamma$. The rules for the two additional judgments are in Figure 4. Intuitively, the rules for *wrap* allow for successively binding the free variables appearing in the method-body (*w_bind*), until it is "closed" (*w_ground*). When we apply the rule (*w_bind*), we choose any (free) variable y in b, and bind it to the corresponding result v, as stated in Γ. The remaining part $\bar{b}$ of the closure can be seen as the closure body of a method where the variable z is supposed to be "closed", and therefore it is obtained in a proof environment containing this information. This is captured by the sub-derivation $\Gamma, closed(z) \vdash wrap(b\{z/y\}, \bar{b}\{z/y\})$, which applies the rule (*w_bind*) until enough variables have been bound and, correspondingly, enough assumptions of the form $closed(z)$ have been taken to be able to prove $closed(b)$ (*i.e.*, there are no more free variables to bind) and thus apply rule (*w_ground*). Notice that the closures we get in this manner are "optimized", because only variables which are really free in the body need to be bound in the closure (although in a non-deterministic order).

The two auxiliary judgments *wrap* and *closed* can be formalized by two inductive predicates, as usual. However, the latter can be managed more efficiently as a function `closed:Term->Prop`, in the style of [24]:

```
Parameter dummy : Var -> Prop.
Fixpoint closed [t:Term] : Prop := Cases t of
    (var x)      => (dummy x) |
                    (obj ml) => (closed_obj ml)
 |  (over a l m) => (closed a) /\
                    ((x:Var) (dummy x)->(closed (m x)))
 |  (call a l)   => (closed a) |
                    (clone a) => (closed a)
 |  (let a b)    => (closed a) /\
                    ((x:Var) (dummy x)->(closed (b x)))
with closed_obj [ml:Object] : Prop := Cases ml of
    (obj_nil)    => True | (obj_cons l m nl) =>
                    (closed_obj nl) /\
                    ((x:Var)(dummy x)->(closed (m x)))
end.
```

The intended behavior of this function, defined by mutual recursion on the structure of terms and objects, is to reduce an assertion `(closed a):Prop` into a conjunction of similar assertions about simpler terms. The `dummy` is the usual workaround for the negative occurrences of `closed` in the definition: dummy variables are just fill-ins for holes, and must be considered as "closed". The proposition resulting from the `Simplification` of a `(closed a)` goal is easily dealt with using the tactics provided by Coq. In a similar way, we define also the functions `notin : Var -> Term -> Prop` and `fresh : Var -> (list Var) -> Prop`, which capture the "freshness" of a variable in a term and *w.r.t.* a list of variables, respectively (see [7]). Finally, the judgment `wrap` is formalized via an inductive predicate:

```
Inductive wrap : Term -> Body -> Prop:=
    w_ground : (b:Term) (closed b) -> (wrap b (ground b))
 |  w_bind   : (b:Var->Term) (c:Var->Body) (y:Var)
               (v:Res) (xl:Varlist)
               ((z:Var) (dummy z) /\
               (fresh z xl)->(wrap (b z) (c z))) ->
               (stack y) = (v) ->
               ((z:Var) ~(y=z)->(notin y (b z))) ->
               (wrap (b y) (bind v c)).
```

In the rule `w_bind`, the premise `((z:Var) (y=z) -> (notin y (b z)))` ensures that `b` is a "good context" for `y`; that is, `y` does not occur free in `b`. Thus, the replacement $b\{z/y\}$ in the rule (*w_bind*) of Figure 4 can be implemented simply as the application `(b z)`, where `z` is the local variable.

All this technical machinery is useful, *e.g.*, in the formalization of the rule ($Red{-}Obj$):

```
e_obj : (s:Store) (ml:Object)
        (cl:(list Closure)) (xl:Varlist)
        (scan (proj_meth_obj (ml)) (cl) (xl)
              (distinct (proj_lab_obj ml))) ->
        (eval s (obj ml) (alloc s cl)
              (new_res_obj ml (size s)))
```

Recall that the reduction of an object `ml` has the side-effect of allocating in the store the collection of closures corresponding to its methods. The `scan` function builds this list of closures `cl` out of an object, using the `wrap` predicate above. Then, the function `alloc` appends the new list of closures to the current store, and the function `new_res_obj` produces a new result, collecting the method-names of the given object and pairing them with fresh pointers to the store. The encoding of the remaining rules is not problematic.

3.3 Static Semantics

Typing of terms. The encoding of the typing system for terms is straightforward. Like done for the reduction context, we model the typing environment by means of a functional symbol, associating object-types to variables:

```
Parameter typenv : Var -> TType.
```

The typing of terms is defined by mutual induction with the typing of objects; notice only that we need to carry along the whole (object) type while we scan and type the list of methods forming the objects:

```
Mutual Inductive type : Term -> TType -> Prop :=
   |   t_sub : (a:Term; A,B:TType) (type a A) ->
               (sub A B) -> (type a B)
```

$$\frac{\Gamma \vdash closed(b)}{\Gamma \vdash wrap(b, ground(b))}\ (w\_ground) \qquad \frac{\Gamma \vdash closed(a)}{\Gamma \vdash closed(a.l)}\ (c\_call) \qquad \frac{\{\Gamma, closed(x_i) \vdash closed(b_i)\}_{i\in I}}{\Gamma \vdash closed([l_i = \varsigma(x_i)b_i]^{i\in I})}\ (c\_obj)$$

$$\frac{\Gamma \vdash closed(a)}{\Gamma \vdash closed(clone(a))}\ (c\_clone) \qquad \frac{\Gamma, closed(z) \vdash wrap(b\{z/y\}, \bar{b}\{z/y\})\ \ z\ fresh\ \ y \mapsto v \in \Gamma}{\Gamma \vdash wrap(b, bind(v, \lambda y.\bar{b}))}\ (w\_bind)$$

$$\frac{\Gamma \vdash closed(a) \quad \Gamma, closed(x) \vdash closed(b)}{\Gamma \vdash closed(a.l \leftarrow \varsigma(x)b)}\ (c\_over) \qquad \frac{\Gamma \vdash closed(a) \quad \Gamma, closed(x) \vdash closed(b)}{\Gamma \vdash closed(let\ x = a\ in\ b)}\ (c\_let)$$

Figure 4: Rules for *wrap* and *closed* judgments.

```
   |   t_obj : (ml:Object) (A:TType)
               (type_obj A (obj ml) A) ->
               (type (obj ml) A) | ...
with type_obj: TType -> Term -> TType -> Prop :=
       t_nil : (A:TType)
               (type_obj A (obj (obj_nil))
                         (mktype (nil (Lab*TType))))
   |   t_cons: (A,B,C:TType; ... m:Var->Term;
                pl:(list (Lab*TType)))
               (type_obj C (obj ml) A) ->
               ((x:Var) (typenv x) = (C) ->
               (type (m x) B)) -> ...
               (type_obj C (obj (obj_cons l m ml))
                         (mktype (cons (l,B) pl))).
```

where `sub` represents the subtype predicate "<:". Due to lack of space, we omit here its encoding, which makes also use of an auxiliary predicate for permutation of lists representing object types (see [7]).

Typing of results. In order to type results, it is necessary to type store-locations, whose content, in turn, may contain other pointers to the store: thus potential loops may arise in this process. The solution adopted in [2] is to introduce yet another typing structure, *i.e. store-types*, used for assigning to each store location a type consistent with its content:

```
Definition SType : Set := (list (TType * TType)).
```

Store-types cannot be inferred from the stores, but have to be provided beforehand; they are used for assigning types to results by means of the auxiliary proof system of Figure 3. Such a system is easily rendered in CIC; we only point out that, similarly to the typing of objects, we have to carry along the whole (result) type while we scan and type the (components of) results:

```
Inductive res : SType -> TType -> Res -> TType -> Prop :=
   t_void : (S:SType) (A:TType)
            (res S A (nil (Lab*Loc)) ->
            (mktype (nil (Lab*TType))))
 | t_step : (S:SType) (A,B,C:TType) (v:Res) (i:Loc)
            (l:Lab) (pl:(list (Lab*TType)))
            (res S A v B) -> (type_from_lab A l)=C ->
            (stype_nth_1 i S) = (A) ->
            (stype_nth_2 i S) = (C) -> ...
            (res S A (cons (l,i) v) ->
            (mktype (cons (l,C) pl))).
Inductive type_body : SType -> Body -> TType -> Prop :=
 t_ground : (S:SType) (b:Term) (A:TType)
            (type b A) -> (type_body S (ground b) A)
 | t_bind : (S:SType) (b:Var->Body) (A,B:TType) (v:Res)
            (type_res S A v A) ->
            ((x:Var) (typenv x) = (A) ->
            (type_body S (b x) B)) ->
            (type_body S (bind v b) B).
```

```
Definition ext : SType -> SType -> Prop := [S',S:SType]
            (le (dim S) (dim S')) /\
            (i:nat) (lt i (dim S)) ->
            (stype_nth i S')=(stype_nth i S).
Definition comp:SType -> Store -> Prop:=[S:SType;s:Store]
            (le (size s) (dim S)) /\
            ((i:nat) (lt i (dim S)) ->
            ((x:Var) (typenv x)=(stype_nth_1 i S) ->
            (type_body S (store_nth i s x)
                         (stype_nth_2 i S)))).
```

The functions for store-type manipulation are in [7].

3.4 Adequacy

Our formalization of the semantics of impς in Coq corresponds faithfully to the NDS one of Section 2, as we see now (for the full proof, we refer the interested reader to [8, 9]). First, we establish the relationship between our heterogeneous (Coq) context Υ and the reduction and typing environments Γ, Δ of the setting of Section 2, and between the two kinds of stores τ and s. In the following, we write $\Upsilon \vdash \mathtt{J}$ for $\exists t.\ \Upsilon \vdash t : \mathtt{J}$.

Definition 2 (Well-formed context). *A context Υ is well-formed if and only if it can be partitioned as $\Upsilon = \Upsilon_s, \Upsilon_t, \Upsilon_d$, where $\Upsilon_s = \ldots, s_i : \mathtt{(stack\ x_i)=v_i}, \ldots$ and $\Upsilon_t = \ldots, t_j : \mathtt{(typenv\ x_j)=A_j}, \ldots$ and $\Upsilon_d = \ldots, d_k : \mathtt{(dummy\ x_k)}, \ldots$; moreover, Υ_s and Υ_t are functional (e.g., if* `stack(x)=v`, `stack(x)=v'` $\in \Upsilon_s$, *then* `v` $\equiv$ `v'`*).*

Definition 3. *Let Υ be a context, Γ, Δ reduction and typing contexts, τ and s stores. We define:*

$$\begin{array}{lcl}
\Upsilon \subseteq \Gamma & \triangleq & \forall \mathtt{(stack\ x)=v} \in \Upsilon.\ x \mapsto v \in \Gamma \\
\Upsilon \subseteq \Delta & \triangleq & \forall \mathtt{(typenv\ x)=A} \in \Upsilon.\ x{:}A \in \Delta \\
\Gamma \subseteq \Upsilon & \triangleq & \forall x \mapsto v \in \Gamma.\ \mathtt{(stack\ x)=v} \in \Upsilon \\
\Delta \subseteq \Upsilon & \triangleq & \forall x{:}A \in \Delta.\ \mathtt{(typenv\ x)=A} \in \Upsilon \\
\gamma(\Gamma) & \triangleq & \{\mathtt{(stack\ x)=}\Gamma(x) \mid x \in \mathsf{Dom}(\Gamma)\} \\
\tau \lesssim s & \triangleq & \forall \iota_i \in \mathsf{Dom}(\tau).\ \gamma(\Gamma_i), \\
 & & \mathtt{(dummy\ x_i)} \vdash \mathtt{(wrap\ b_i\ ((}\tau\ \iota_i\mathtt{)\ x_i))}, \\
 & & \textit{where } s(\iota_i) = \langle \varsigma(x_i)b_i, \Gamma_i \rangle \\
s \lesssim \tau & \triangleq & \forall \iota_i \in \mathsf{Dom}(s).\ \gamma(\Gamma_i), \\
 & & \mathtt{(dummy\ x_i)} \vdash \mathtt{(wrap\ b_i\ ((}\tau\ \iota_i\mathtt{)\ x_i))}, \\
 & & \textit{where } s(\iota_i) = \langle \varsigma(x_i)b_i, \Gamma_i \rangle
\end{array}$$

In the following, for $\bar{b}$ a closure-body, let us denote by $stack(\bar{b})$ the reduction context containing the bindings in $\bar{b}$, and by $body(\bar{b})$ the innermost body. These functions can be defined

by recursion:

$$\begin{array}{lcl} stack(ground(b)) & \triangleq & \emptyset \\ stack(bind(v, \lambda x.\bar{b})) & \triangleq & stack(\bar{b}) \cup \{x \mapsto v\} \\ body(ground(b)) & \triangleq & b \\ body(bind(v, \lambda x.\bar{b})) & \triangleq & body(\bar{b}) \end{array}$$

THEOREM 2 (ADEQUACY OF REDUCTION). *Let Υ be well-formed, Γ a reduction context.*

1. *Let $\Upsilon \subseteq \Gamma$, and $\tau \lesssim s$.*
   (a) $\Upsilon \vdash$ `(eval` τ `a` τ' `v)` $\Rightarrow \exists s'$. $\Gamma \vdash s \cdot a \rightsquigarrow v \cdot s'$, *and* $\tau' \lesssim s'$;
   (b) $\Upsilon \vdash$ `(eval_body` τ $\bar{b}$ τ' `v)` $\Rightarrow \exists s'$. $stack(\bar{b}) \vdash s \cdot body(\bar{b}) \rightsquigarrow_b v \cdot s'$, *and* $\tau' \lesssim s'$.
2. *Let $\Gamma \subseteq \Upsilon$, and $s \lesssim \tau$. $\Gamma \vdash s \cdot a \rightsquigarrow v \cdot s' \Rightarrow \exists \tau'$. $\Upsilon \vdash$ `(eval` τ `a` τ' `v)`, and $s' \lesssim \tau'$.*

PROOF. 1. The two points are proved by mutual structural induction on the derivations of $\Upsilon \vdash$ `(eval` τ `a` τ' `v)` and $\Upsilon \vdash$ `(eval_body` τ $\bar{b}$ τ' `v)`.

2. By structural induction on the derivation of $\Gamma \vdash s \cdot a \rightsquigarrow v \cdot s'$.

□

THEOREM 3 (ADEQUACY OF TERM TYPING). *Let Υ be well-formed, Δ a typing context.*

1. *If $\Upsilon \subseteq \Delta$, and $\Upsilon \vdash$ `(type a A)`, then $\Delta \vdash a : A$.*
2. *If $\Delta \subseteq \Upsilon$, and $\Delta \vdash a : A$, then $\Upsilon \vdash$ `(type a A)`.*

PROOF. 1. By structural induction on the derivation of $\Upsilon \vdash$ `(type a A)`.

2. By structural induction on the derivation of $\Delta \vdash a : A$.

□

THEOREM 4 (ADEQUACY OF RESULT TYPING). *Let Υ be well-formed, and Σ such that $\Sigma \vdash \diamond$.*

1. *For $\tau \lesssim s$, if $\Upsilon \vdash$ `(res` Σ `v A)` and $\Upsilon \vdash$ `(comp` Σ τ`)`, then $\Sigma \models v : A$ and $\Sigma \models s$.*
2. *For $s \lesssim \tau$, if $\Sigma \models v : A$ and $\Sigma \models s$, then $\Upsilon \vdash$ `(res` Σ `v A)` and $\Upsilon \vdash$ `(comp` Σ τ`)`.*

PROOF. 1. By induction on v and inspection on the derivation $\Upsilon \vdash$ `(comp` Σ τ`)`.

2. By inspection on the derivations of $\Sigma \models v : A$ and $\Sigma \models s$.

□

4. METATHEORY OF IMPς IN Coq

One of the main aims of the formalization presented in the previous section is to allow for the formal development of important properties of impς. In this section we discuss briefly the uppermost important, yet delicate to prove, metaproperty of the Subject Reduction (Theorem 1). Its formalization in Coq is as follows:

```
Theorem SR: (s,t:Store) (a:Term) (v:Res)
            (eval s a t v) -> (A:TType)(type a A) ->
            (S:SType)(comp S s) ->
            ((x:Var;w:Res;C:TType) (stack x) = (w) /\
            (typenv x) = C -> (res S w C)) ->
            (EX B:TType | (EX T:SType |
                          (res T B v B) /\ (ext T S) /\
                          (comp T t) /\ (sub B A))).
```

The proof is by induction on the derivation of `(eval s a t v)`. In order to prove the theorem, we have to address preliminarly all the specific aspects concerning the management of concrete structures, as objects, object-types, results, stores, store-types, and so on: thus, many technical lemmata about operational semantics, term and result typing have been specified and formally proved. It turns out that these lemmata are relatively compact to prove, due to the setting of our formalization, which is centered around the natural deduction style of proof and, correspondingly, the use of HOAS.

Natural Deduction allows to distribute stack-like structures (*i.e.* reduction and typing contexts) in the hypotheses of the proof derivations: therefore, judgments and proofs we have to deal with are appreciably simpler than traditional ones. Weak HOAS, as previously remarked, fits perfectly the needs for encoding impς: on one hand, it allows to get the α-equivalence for free; on the other hand, the fact that the general form of substitution is not automatically provided by weak HOAS is not a problem, because it is not required by the semantics of impς.

However, it is well-known that HOAS presents some drawbacks: the main one is that most LFs do not provide an adequate support for higher-order encodings. For example, these systems neither provide recursion/induction principles over higher-order terms (*i.e.*, terms with "holes") nor allow to access the notions related to the mechanisms delegated to the metalanguage. An important family of properties which cannot be proved in plain CIC are the so-called *renaming lemmata*, that is, invariance of validity under variable renaming, such as the following, regarding types of terms and closure-bodies of impς:

```
Lemma rename_term : (m:Var->Term) (A:TType) (x,y:Var)
                    (type (m x) A) ->
                    (typenv x) = (typenv y) ->
                    (type (m y) A).
Lemma rename_body : (S:SType) (c:Var->Body)
                    (A:TType) (x,y:Var)
                    (type_body S (c x) A) ->
                    (typenv x) = (typenv y) ->
                    (type_body S (c y) A).
```

In other words, the expressive power of LFs is limited, when it comes to reason on formalizations in (weak) HOAS.

A simple and direct way for recovering the missing expressive power is by assuming a suitable (but consistent) set of *axioms*. This is the approach adopted in [17, 28], where ToC, a simple axiomatization capturing some basic and natural properties of *(variable) names* and *term contexts*, is proposed. These axioms allow for a smooth handling of schemata in HOAS, with a very low mathematical and logical overhead, hence they can be plugged in existing proof environments without requiring any redesign of the system. Their usefulness has been demonstrated in several case studies regarding untyped and simply typed λ-calculus, π-calculus, and Ambients [18, 23, 29]. Therefore, the use of the ToC is a natural choice also in the present setting; it should be noticed that the present one is the first application of the ToC to an object-based calculus. The Theory of Contexts consists in four axioms (indeed, axiom schemata):

unsaturation: $\forall M.\ \exists x.\ x \notin \mathsf{FV}(M)$. This axiom captures the intuition that, since terms are finite entities, they cannot contain all the variables at once. The same axiom can be stated *w.r.t.* lists of variables, instead of terms, since it is always possible to obtain from a term the list of its free variables: $\forall L.\ \exists x.\ x \notin L$;

decidability of equality over variables: $\forall x, y.\ x = y \vee x \neq y$. In a classical framework, this axiom is just an instance of the *Law of Excluded Middle*. In the present case, it represents the minimal classical property we need in an (otherwise) intuitionistic setting;

β-expansion: $\forall M, x.\ \exists N(\cdot).\ x \notin \mathsf{FV}(N(\cdot)) \wedge M = N(x)$. Essentially, β-expansion allows to generate a new context N with one more hole from another context M, by abstracting over a given variable x;

extensionality: $\forall M(\cdot), N(\cdot), x.\ x \notin \mathsf{FV}(M(\cdot), N(\cdot)) \wedge (M(x) = N(x)) \Rightarrow M(\cdot) = N(\cdot)$. Extensionality allows to conclude that two contexts are equal if they are equal when applied to a fresh variable x. Together with β-expansion, this allows to derive properties by reasoning over the structure of higher-order terms.

The above axioms are very natural and useful for dealing with higher-order terms as methods, closures and local declarations. An important remark is that, in order to be effectively used for reasoning on impς, the "unsaturation" axiom has to be slightly modified with respect to its original formulation in [17].

One first difference is due to the presence of types. Similarly to the case of other typed languages, we assume informally that there are infinite variables for every type. This is equivalent to say that each variable "generated" by the unsaturation axiom can be associated to a given type.

A second difference, peculiar to impς and not required by any previous application of the ToC, is due to the presence of implementation-level entities, namely *closures*. In the formalization of impς presented in the previous section, variables are introduced in the Coq derivation context for two reasons: either during reductions and typing assignments (associated respectively to results and types), or in the construction of closures (used just as place-holders). In the first case the new variable is associated both to results and types by the `stack` and `typenv` maps. In the second case, the new variable is marked as `dummy`, because it does not carry any information about results, but is used just as a typed place-holder, needed for typing closure-bodies in the store.

Thus, we observe a kind of "regularity" of well-formed contexts: for each variable x, there is always the assumption `(typenv x)=A` for some `A`, and, either `(stack x)=v` for some `v`, or `(dummy x)`. The unsaturation axiom has to respect such a regularity; this is reflected by assuming two variants of unsaturation, one for each case, as follows:

```
Axiom unsat_res : (S:SType) (v:Res) (A:TType)
   (res S v A) -> (xl:(list Var))
   (EX x | (fresh x xl) /\ (stack x)=v /\ (typenv x)=A).
Axiom unsat : (A:TType) (xl:list Var)
   (EX x | (fresh x xl) /\ (dummy x) /\ (typenv x)=A).
```

In `unsat_res`, the premise `(res S v A)` ensures the consistency between results and types associated to the same variable: this can be interpreted as the implementation of the original context compatibility judgment of Figure 3.

The two axioms can be validated in models similar to those of the original ToC [3]. Moreover, the notion of regularity is close to the "regular world assumption" introduced by Schürmann [30].

In order to convey better to the reader the above explanation about the ToC, we say that a typical example of the use of the axiom `unsat` is for proving that the type of a closure-body is preserved by the closure construction:

```
Lemma wrap_pres_type: (A,B:TType; m:Var->Term; c:Closure;
                       xl:(list Var); S:SType)
      ((x:Var) (typenv x)=A -> (type (m x) B)) ->
      ((x:Var) (dummy x) /\
               (fresh x xl) -> (wrap (m x) (c x))) -> ...
      ((x:Var) (typenv x)=A -> (type_body S (c x) B)).
```

provided the maps `stack` and `typenv` are consistent *w.r.t.* the store-type S. The proof of this property requires the use of both `unsat` and the decidability of equality over variables. On the other hand, the proof of the lemmata `rename_term` and `rename_body`, previously displayed, requires the application of the β-expansion and extensionality.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have illustrated the benefits of using Natural Deduction in combination with Higher-Order Abstract Syntax and the Theory of Contexts for reasoning on object-calculi with binders in type theory-based logical frameworks. We have carried out our experiment on Abadi and Cardelli's impς, an object-based calculus featuring types and side-effects. Natural Deduction style of proof has allowed to distribute in the hypotheses of proof derivations both the reduction and typing context of impς, thus obtaining judgments and proofs appreciably simpler than traditional ones. Weak HOAS has permitted to deal with the binders of impς in the Calculus of Inductive Constructions without having to encode neither α-equivalence (which is inherited from the metalanguage) nor the substitution (which is not required by the calculus). This is a big advantage from the point of view of computer aided formal reasoning *w.r.t.* first-order techniques, as de Bruijn indexes or explicit names. A consequence of our choices is that we have obtained a more fine-grained treatment of closures. Finally, for reasoning on the formalization of impς in CIC we have adopted the Theory of Contexts, in order to gain the extra power CIC needs for reasoning on HOAS encodings.

Thus we have obtained a clean and compact formalization of impς in the proof assistant Coq, the implementation of CIC. Our style of encoding has allowed to prove formally a Subject Reduction theorem, which is particularly involved already "on paper", with relatively little effort. Just notice that the full proof development amounts approximately to $112Kbyte$ and the size of the `.vo` file is $785Kbyte$, working with Coq V7.3 on a Sun UltraSPARC IIe (64 bit).

To our knowledge, this is the first development of the theory of an object-based language with side effects, in LF based on type theory. The closest work may be [21], where Abadi and Cardelli's *functional* object-based calculus $Ob_{1<:\mu}$ is encoded in Coq, using traditional first-order techniques and Natural Semantics specifications through the Centaur system. A logic for reasoning *on paper* about object-oriented programs with imperative semantics, aliasing and self-reference in objects, has been presented in [1].

The Theory of Contexts has been already used with the weak HOAS for carrying out many case studies in recent years (see, *e.g.*, [18, 19, 23, 29]); however, the present work is the first application of the ToC to an object calculus. Our experience leads us to affirm that this approach is particularly well-suited with respect to the proof practice of Coq. The ToC can be actually plugged in existing LFs without requiring any redesign of these systems. In particular, it seems very suited for dealing with implementation-level structures, such as closures, since full substitution is not required.

Future work. As a first step, we plan to experiment further with the formalization we have carried out so far. We will consider other interesting (meta)properties of impς, beside the albeit fundamental Subject Reduction theorem. In particular, we can use the formalization for proving observational and behavioral equivalences of object programs.

From a practical point of view, our formalization could be used for the development of *certified* tools, such as interpreters, compilers and type checkers, for impς-like calculi and languages. We plan to certify a given tool with respect to the specification of the formal semantics of the object calculus and the target machine. We are confident that the use of Natural Deduction and HOAS should simplify these advanced tasks in the case of languages with binders.

From a more theoretical point of view, we had to modify slightly the unsaturation axiom of the ToC. This has occurred also in other applications of ToC [23, 29]. From all these case studies, we observe that these adaptments are required when the proof contexts have to satisfy some kind of "well-formedness" about variables: every time a new variable is generated, it has to come together with a set of other assumptions and informations (about its type, its kind, etc.). This points out that the current ToC can be generalized to subsume all these small variants, and that there is a connection with Schürmann's *regular world assumption* [30], which should be investigated further.

6. REFERENCES

[1] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In *Proc. of TAPSOFT*, *LNCS* 1214, pages 682–696. Springer-Verlag, 1997.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

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

[4] R. Burstall and F. Honsell. Operational semantics in a natural deduction setting. In *Logical Frameworks*, pages 185–214. CUP, 1990.

[5] L. Cardelli. Obliq: A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[6] A. Ciaffaglione, L. Liquori, and M. Miculan. Formalization of imperative object-calculi in (co)inductive type theories. Submitted, 2003.

[7] A. Ciaffaglione, L. Liquori, and M. Miculan. The Web Appendix of this paper, 2003. `http://www.dimi.uniud.it/~ciaffagl/Objects/Imp-varsigma.tar.gz`.

[8] A. Ciaffaglione. *Certified reasoning on Real Numbers and Objects in Co-inductive Type Theory*. PhD thesis, DiMI, Univ. di Udine and LORIA-INPL, Nancy, 2003.

[9] Alberto Ciaffaglione, Luigi Liquori, and Marino Miculan. On the formalization of imperative object-based calculi in (co)inductive type theories. Research Report RR-4812, INRIA, 2003.

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

[11] J. Despeyroux and P. Leleu. Primitive recursion for higher-order abstract syntax with dependant types. In *FLoC IMLA workshop*, 1999.

[12] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In [20], pages 193–202.

[13] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1994.

[14] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 15:341–363, 2002.

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

[16] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. of LICS* [20], pages 204–213.

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

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

[19] F. Honsell, M. Miculan, and I. Scagnetto. The theory of contexts for first-order and higher-order abstract syntax. In *Proc. of TOSCA*, *ENTCS* 62, pages 111–130. Elsevier, 2001.

[20] G. Longo, editor. *Proc. of LICS*, IEEE, 1999.

[21] O. Laurent. Sémantique Naturelle et Coq : vers la spécification et les preuves sur les langages à objets. Rapport de Recherche RR-3307, INRIA, 1997.

[22] M. Miculan. The expressive power of structural operational semantics with explicit assumptions. In *Proc. of TYPES*, *LNCS* 806, pages 292–320, Springer-Verlag, 1994.

[23] M. Miculan. Developing (meta)theory of λ-calculus in the Theory of Contexts. In *Proc. of MERLIN*, *ENTCS* 58.1, pages 1–22. Elsevier, 2001.

[24] M. Miculan. On the formalization of the modal μ-calculus in the Calculus of Inductive Constructions. *Information and Computation*, 164(1):199–231, 2001.

[25] A. Momigliano and S. J. Ambler. Multi-level meta-reasoning with higher order abstract syntax. In *Proc. of FoSSaCS*, *LNCS* 2620, pages 375–391. Springer-Verlag, 2003.

[26] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. of SLDI*, pages 199–208, ACM Press, 1988.

[27] 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 *Proc. of FOSSACS*, *LNCS* 2030, pages 359–373, Springer-Verlag, 2001

[28] I. Scagnetto. *Reasoning about Names In Higher-Order Abstract Syntax*. PhD thesis, DiMI, Università di Udine, Italy, 2002.

[29] I. Scagnetto and M. Miculan. Ambient calculus and its logic in the calculus of inductive constructions. In *Proc. of LFM*, *ENTCS* 70.2. Elsevier, 2002.

[30] C. Schürmann. Recursion for higher-order encodings. In *Proc. of CSL*, *LNCS* 2142, pages 585-599. Springer-Verlag, 2001.