



UNIVERSITÀ
DEGLI STUDI
DI UDINE

Università degli studi di Udine

Imperative Object-based Calculi in (Co)Inductive Type Theories

Original

Availability:

This version is available <http://hdl.handle.net/11390/739889>

since 2016-11-24T12:53:04Z

Publisher:

Published

DOI:10.1007/978-3-540-39813-4_4

Terms of use:

The institutional repository of the University of Udine (<http://air.uniud.it>) is provided by ARIC services. The aim is to enable open access to all the world.

Publisher copyright

(Article begins on next page)

Imperative Object-based Calculi in Co-Inductive Type Theories^{*}

Alberto Ciaffaglione¹, Luigi Liquori², and Marino Miculan¹

¹DiMI, Università di Udine, Italy

²INRIA-LORIA, Nancy, France

[ciaffagl,miculan]@dimi.uniud.it lliquori@loria.fr

Abstract. We discuss the formalization of Abadi and Cardelli’s $\text{imp}_{\mathcal{C}}$, a paradigmatic object-based calculus with types and side effects, in Co-Inductive Type Theories, such as the *Calculus of (Co)Inductive Constructions* ($\text{CC}^{(\text{Co})\text{Ind}}$).

Instead of representing directly the original system “as it is”, we reformulate its syntax and semantics bearing in mind the proof-theoretical features provided by the target metalanguage. On one hand, this methodology allows for a smoother implementation and treatment of the calculus in the metalanguage. On the other, it is possible to see the calculus from a new perspective, thus having the occasion to suggest original and cleaner presentations.

We give hence a new presentation of $\text{imp}_{\mathcal{C}}$, exploiting *natural deduction semantics*, (*weak*) *higher-order abstract syntax*, and, for a significant fragment of the calculus, *coinductive* typing systems. This presentation is easier to use and implement than the original one, and the proofs of key metaproperties, *e.g.* subject reduction, are much simpler.

Although all proof developments have been carried out in the Coq system, the solutions we have devised in the encoding of and metareasoning on $\text{imp}_{\mathcal{C}}$ can be applied to other imperative calculi and proof environments with similar features.

Introduction

In recent years, much effort has been put in the formalization of class-based object-oriented languages. The Coq system [19] has been used for studying formally the JavaCard Virtual Machine and its platform [4], and for checking the behavior of a byte-code verifier for the JVM language [6]. PVS and Isabelle have been used for formalizing and certifying an executable bytecode verifier for a significant subset of the JVM [21], for reasoning on Java programs with Hoare-style logics [18] and on translations of coalgebraic specifications to programs in JavaCard [29] and C++ [28].

In spite of this large contribution on class-based languages, relatively little or no formal work exists for *object-based* ones, like *e.g.* Self and Obliq, where there is no notion of “class” and objects may act as prototypes. This is due mainly to the fact that object-based languages are less used in practice than class-based ones. However, the former are simpler and provide more primitive and flexible mechanisms, and can be used as intermediate level for implementing the latter. From a foundational point of view, indeed, most of the calculi introduced for the mathematical analysis of the object-oriented paradigm are object-based, as *e.g.* [1, 14]. Among the several calculi, Abadi and Cardelli’s $\text{imp}_{\mathcal{C}}$ [1] is particularly representative: it features objects, methods, cloning, dynamic lookup, method update, types, subtypes, and, last but not least, imperative features. This makes $\text{imp}_{\mathcal{C}}$ quite complex, both at the syntactic and at the semantic level. Beside the idiosyncrasies of functional languages with imperative features, the

^{*} Research supported by Italian MIUR project COFIN 2001013518 CoMETA.

store model underlying imp_ζ allows for loops, thus making the typing system quite awkward. This level of complexity is reflected in developing metatheoretic properties; for instance, the fundamental *subject reduction* and *type soundness* are much harder to state and prove for imp_ζ than for traditional functional languages.

It is clear that this situation can benefit from the use of *proof assistants*, where the theory of an object system can be formally represented in some metalanguage, the proofs can be checked, and new, error-free proofs can be safely developed in interactive sessions. However, up to our knowledge, there is no formalization of an object-based calculus with side-effects like imp_ζ , yet. This is, in fact, the aim of our work.

In this paper, we represent and reason on both static and dynamic aspects of imp_ζ in a proof assistant based on type theory, *i.e.* Coq. To this end we will use Coq’s specification language, the coinductive type theory $\text{CC}^{(\text{Co})\text{Ind}}$, as a Logical Framework (LF). The encoding methodology forces us to spell out in full detail all aspects of the calculus, thus any problematic issues which are skipped on paper are identified and fixed. Moreover, we have the occasion to (re)formulate the object system, taking full advantage of the definition and proof-theoretical principles provided by the LF, whose perspective may suggest alternative, and cleaner, definitions of the same systems. In particular, most type theory-based LFs support *natural deduction* and (weak) *higher-order abstract syntax*, and some of them even *coinductive* datatypes and predicates, as in the case of $\text{CC}^{(\text{Co})\text{Ind}}$.

Therefore, we reformulate the static and dynamic semantics of imp_ζ in the style of *Natural Deduction Semantics* (NDS) [7, 23] (the counterpart in Natural Deduction of Kahn’s *Natural Semantics* [12, 20]) using weak higher-order abstract syntax. In this way, α -conversion and the handling of structures which obey a stack discipline (such as the *environments*), are fully delegated to the metalanguage, making judgments and proofs rather simpler than traditional ones.

Another key proof-theoretical innovation is the use of *coinductive* types and proof systems. This is motivated by the observation that the proof of the Subject Reduction in [1] is quite involved, mainly because the store may contain “pointer loops”. Since loops have a non well-founded nature, usual inductive arguments cannot be applied and extra structures, the so-called *store types*, have to be introduced and dealt with. However, coinductive tools are seen nowadays as the canonical way for dealing with circular, non well-founded objects. Therefore, we elaborate a novel and original coinductive reformulation of the typing system for the fragment of imp_ζ without *method update* (which we denote by imp_ζ^-), thus getting rid of the extra structure of store types and making the proof of the Subject Reduction dramatically simpler. It is still an open question whether our coinductive approach can be extended to the full imp_ζ , which has been formalized, at the moment, using store types and related typing system (but still using HOAS and NDS). Due to lack of space, we discuss in this paper only the imp_ζ^- fragment; we refer to [10] for the treatment of the full imp_ζ .

Our effort is useful also from the point of view of LFs. The theoretical development of LFs and their implementation will benefit from complex case studies like the present one, where we test the applicability of advanced encoding methodologies. In this perspective, our contribution can be considered pioneering in combining the higher-order approach with coinductive proof systems in natural deduction style. The techniques we have developed in the encoding of and metareasoning on imp_ζ can be reused for other imperative calculi featuring similar issues.

$$\begin{array}{c}
\frac{}{\emptyset \vdash \diamond} \text{ (Store-}\emptyset\text{)} \qquad \frac{\sigma \cdot S \vdash \diamond \quad \iota \notin \text{Dom}(\sigma)}{\sigma, \iota \mapsto \langle \varsigma(x)b, S \rangle \vdash \diamond} \text{ (Store-}\iota\text{)} \\
\\
\frac{\sigma \vdash \diamond}{\sigma \cdot \emptyset \vdash \diamond} \text{ (Stack-}\emptyset\text{)} \qquad \frac{\sigma \cdot S \vdash \diamond \quad \iota_i \in \text{Dom}(\sigma) \quad x \notin \text{Dom}(S) \quad \forall i \in I}{\sigma \cdot (S, x \mapsto [l_i = \iota_i]^{i \in I}) \vdash \diamond} \text{ (Stack-Var)}
\end{array}$$

Fig. 1. Well-formedness for Store and Stack

Synopsis. Section 1 gives a brief account of imp_{ς} . In Section 2 we focus on the fragment $\text{imp}_{\varsigma}^{-}$, which is reformulated bearing in mind the proof-theoretical concepts provided by $\text{CC}^{(\text{Co})\text{Ind}}$. The formalization in Coq of this system, and the formal proof of the Subject Reduction, are discussed in Sections 3 and 4, respectively. Conclusions and directions for future work are presented in Section 5. The Coq code is available at [11].

1 Abadi and Cardelli's imp_{ς} Calculus

The imp_{ς} -calculus is an imperative calculus of objects forming the kernel of the language Obl_{iq} [8]. The syntax of imp_{ς} is the following:

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

Notice that let and ς bind x in b , and that usual conventions about α -conversion apply. We refer to [1, Ch.10,11] for an explanation of the intuitive meaning of these constructs.

Dynamic Semantics. The big-step operational semantics is expressed by a reduction relation relating a store σ , a stack S , a term a , a result v and another store σ' , *i.e.* $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$. The intended meaning is that, with the store σ and the stack S , the term a reduces to a result v , yielding an updated store σ' and leaving the stack S unchanged in the process. The sorts involved in the reduction semantics are the following:

$Loc : \iota \in \text{Nat}$	store location	$Stack : S ::= x_i \mapsto v_i^{i \in I}$	stack
$Res : v ::= [l_i = \iota_i]^{i \in I}$	result	$Store : \sigma ::= \iota_i \mapsto \langle \varsigma(x_i)b_i, S_i \rangle^{i \in I}$	store

A *store* is a function mapping locations to *method closures*: closures (denoted by c) are pairs built of *methods* and *stacks*. Stacks are used for the reduction of the method bodies: they associate variables with *object results*. Results are sequences of pairs: method labels together with store locations, one location for each object method. The operational semantics needs two auxiliary judgments, namely $\sigma \vdash \diamond$, and $\sigma \cdot S \vdash \diamond$ (Figure 1), checking the well-formedness of stores and stacks, respectively. 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 $i \in I$; the store $\sigma, \iota \mapsto c$ extends σ with c at ι (fresh) and $\sigma, \iota_j \leftarrow c$ denotes the result of replacing the content of the location ι_j of σ with c . Unless explicitly remarked, all the l_i, ι_i are distinct. The rules for the reduction judgment are in Figure 2.

Static Semantics. The type system is first-order with subtyping. The only type constructor is the one for object types, *i.e.* $TType : A, B ::= [l_i : A_i]^{i \in I}$, so the only ground type is $[]$. The typing environment E consists of a list of assumptions for

$$\begin{array}{c}
\frac{\sigma \cdot (S', x \mapsto v, S'') \vdash \diamond}{\sigma \cdot (S', x \mapsto v, S'') \vdash x \rightsquigarrow v \cdot \sigma} \text{ (Red-Var)} \\
\\
\frac{\sigma \cdot S \vdash a \rightsquigarrow v' \cdot \sigma' \quad \sigma' \cdot (S, x \mapsto v') \vdash b \rightsquigarrow v'' \cdot \sigma''}{\sigma \cdot S \vdash \text{let } x = a \text{ in } b \rightsquigarrow v'' \cdot \sigma''} \text{ (Red-Let)} \\
\\
\frac{\sigma \cdot S \vdash \diamond \quad \iota_i \notin \text{Dom}(\sigma) \quad \forall i \in I}{\sigma \cdot S \vdash [l_i = \varsigma(x_i)b_i]^{i \in I} \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot (\sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, S \rangle^{i \in I})} \text{ (Red-Obj)} \\
\\
\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot \sigma' \quad \iota_i \in \text{Dom}(\sigma') \quad \iota'_i \notin \text{Dom}(\sigma') \quad \forall i \in I}{\sigma \cdot S \vdash \text{clone}(a) \rightsquigarrow [l_i = \iota'_i]^{i \in I} \cdot (\sigma', \iota'_i \mapsto \sigma'(\iota_i)^{i \in I})} \text{ (Red-Clone)} \\
\\
\frac{\begin{array}{l} \sigma'(\iota_j) = \langle \varsigma(x_j)b_j, S' \rangle \quad x_j \notin \text{Dom}(S') \quad j \in I \\ \sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot \sigma' \quad \sigma' \cdot (S', x_j \mapsto [l_i = \iota_i]^{i \in I}) \vdash b_j \rightsquigarrow v \cdot \sigma'' \end{array}}{\sigma \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma''} \text{ (Red-Sel)} \\
\\
\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot \sigma' \quad \iota_j \in \text{Dom}(\sigma') \quad j \in I}{\sigma \cdot S \vdash a.l_j \leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i]^{i \in I} \cdot (\sigma'.l_j \leftarrow \langle \varsigma(x)b, S \rangle)} \text{ (Red-Upd)}
\end{array}$$

Fig. 2. Natural Operational Semantics for imp_{ς}

variables, each of the form $x:A$. The type system is given by four judgments: well-formedness of type environment $E \vdash \diamond$, well-formedness of object types $E \vdash A$, subtyping $E \vdash A <: B$ and term typing $E \vdash a : A$. Rules for these judgments are collected in Figures 3 and 4. Notice that the subtype relation between object types induces the notion of *subsumption*: an object of a given type also belongs to any supertype of that type and can subsume objects in the supertype, because these have a more limited protocol. The rule (*Sub-Obj*) allows a longer object type to be a subtype of a shorter one: $[l_i : A_i]^{i \in I \cup J} <: [l_i : B_i]^{i \in I}$ requires $A_i \equiv B_i$ for all $i \in I$; that is, object types are *invariant* (i.e. neither covariant nor contravariant) in their component types. This condition guarantees the soundness of the type discipline.

Result and store typing. The typing of results is delicate, because results point to the store, and stores may contain loops: thus it is not possible to determine the type of a result examining its substructures recursively. *Store types* allow to type results independently of particular stores: this is possible because type-sound computations do not store results of different types in the same location. A store type Σ 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 result type:

$$\begin{array}{ll}
M ::= [l_i : B_i]^{i \in I} \Rightarrow B_j \quad (j \in I) & \Sigma_1(\iota) \triangleq [l_i : B_i]^{i \in I} \text{ if } \Sigma(\iota) = [l_i : B_i]^{i \in I} \Rightarrow B_j \\
\Sigma ::= \iota_i \mapsto M_i^{i \in I} & \Sigma_2(\iota) \triangleq B_j \quad \text{if } \Sigma(\iota) = [l_i : B_i]^{i \in I} \Rightarrow B_j
\end{array}$$

The system for store typing is given by five judgments: well-formedness of method types $M \models \diamond$ and store types $\Sigma \models \diamond$, result typing $\Sigma \models v : A$, store typing $\Sigma \models \sigma$,

$$\begin{array}{c}
\frac{}{\emptyset \vdash \diamond} (Env-\emptyset) \qquad \frac{E \vdash A \quad x \notin \text{Dom}(E)}{E, x:A \vdash \diamond} (Env-Var) \\
\\
\frac{E \vdash A_i \quad \forall i \in I}{E \vdash [l_i : A_i]^{i \in I}} (Type-Obj) \qquad \frac{E \vdash A}{E \vdash A <: A} (Sub-Ref1) \\
\\
\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C} (Sub-Trans) \qquad \frac{E \vdash A_i \quad \forall i \in I \cup J}{E \vdash [l_i : A_i]^{i \in I \cup J} <: [l_i : A_i]^{i \in I}} (Sub-Obj)
\end{array}$$

Fig. 3. Auxiliary Typing judgments

$$\begin{array}{c}
\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B} (Val-Sub) \qquad \frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x : A} (Val-Var) \\
\\
\frac{E, x_i:[l_i : A_i]^{i \in I} \vdash b_i : A_i \quad \forall i \in I}{E \vdash [l_i = \varsigma(x_i)b_i]^{i \in I} : [l_i : A_i]^{i \in I}} (Val-Obj) \qquad \frac{E \vdash a : [l_i : A_i]^{i \in I} \quad j \in I}{E \vdash a.l_j : A_j} (Val-Sel) \\
\\
\frac{E \vdash a : [l_i : A_i]^{i \in I}}{E \vdash \text{clone}(a) : [l_i : A_i]^{i \in I}} (Val-Clone) \qquad \frac{E \vdash a : A \quad E, x:A \vdash b : B}{E \vdash \text{let } x = a \text{ in } b : B} (Val-Let) \\
\\
\frac{E \vdash a : [l_i : A_i]^{i \in I} \quad E, x:[l_i : A_i]^{i \in I} \vdash b : A_j \quad j \in I}{E \vdash a.l_j \leftarrow \varsigma(x)b : [l_i : A_i]^{i \in I}} (Val-Upd)
\end{array}$$

Fig. 4. Type Checker for imp_{ς}

and stack typing $\Sigma \models S : E$, which are given in Figure 5. The intended meaning of the judgment $\Sigma \models v : A$ is to assign the type A to the result v looking at Σ , and the judgment $\Sigma \models \sigma$ ensures that the content of every store location of σ is given the type of the same location of Σ .

Subject Reduction. An important property of imp_{ς} is 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 (see [1] for a complete proof).

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

Theorem 1 (Subject Reduction). If $E \vdash a : A$, and $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$, and $\Sigma \models \sigma$, and $\text{Dom}(\sigma) = \text{Dom}(\Sigma)$, and $\Sigma \models S : E$, then there exist a type A' , and a store type Σ' , such that $A' <: A$, $\Sigma' \geq \Sigma$, $\Sigma' \models \sigma'$, $\text{Dom}(\sigma') = \text{Dom}(\Sigma')$, and $\Sigma' \models v : A'$.

2 imp_{ς}^- in Coinductive Natural Deduction Semantics

In this section, we focus on the fragment of imp_{ς} without method update, denoted by imp_{ς}^- . For this fragment, we elaborate an original, alternative presentation of static and

$$\begin{array}{c}
\frac{j \in I}{[l_i : B_i]^{i \in I} \Rightarrow B_j \models \diamond} \text{ (Meth-Type)} \\
\\
\frac{\Sigma \models \diamond \quad \Sigma_1(\iota_i) = [l_i : \Sigma_2(\iota_i)]^{i \in I} \quad \forall i \in I}{\Sigma \models [l_i = \iota_i]^{i \in I} : [l_i : \Sigma_2(\iota_i)]^{i \in I}} \text{ (Res)} \\
\\
\frac{\Sigma \models \diamond}{\Sigma \models \emptyset : \emptyset} \text{ (Stack-}\emptyset\text{-Typ)} \\
\\
\frac{M_i \models \diamond \quad \forall i \in I}{\iota_i \mapsto M_i^{i \in I} \models \diamond} \text{ (Store-Type)} \\
\\
\frac{\Sigma \models S_i : E_i \quad \forall i \in I \quad E_i, x_i : \Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\iota_i)}{\Sigma \models \iota_i \mapsto \langle \varsigma(x_i)b_i, S_i \rangle^{i \in I}} \text{ (Store-Typing)} \\
\\
\frac{x \notin \text{Dom}(S, E) \quad \Sigma \models S : E \quad \Sigma \models v : A}{\Sigma \models S, x \mapsto v : E, x : A} \text{ (Stack-Var-Typ)}
\end{array}$$

Fig. 5. Extra judgments for Store Types

dynamic semantics, taking advantage of $\text{CC}^{(\text{Co})\text{Ind}}$, a type theory providing *natural deduction*, *higher-order abstract syntax* and *coinductive types*. This setting, which we call *Coinductive Natural Deduction Semantics*, leads to a very clean and compact system, allowing for an easier treatment of theoretical and metatheoretical results. The major changes of our reformulation concern the operational semantics, whereas the type system for terms needs only a minor revision; very delicate is the typing of results, which makes also use of coinductive principles.

Syntax. Following the higher-order abstract syntax paradigm [16, 25], we reduce all binders to the sole λ -abstraction. Therefore, from now on we write $\text{let}(a, \lambda x.b)$ for $\text{let } x = a \text{ in } b$, and $\varsigma(\lambda x.b)$ for $\varsigma(x)b$, where $\text{let} : \text{Term} \times (\text{Var} \rightarrow \text{Term}) \rightarrow \text{Term}$, and $\varsigma : (\text{Var} \rightarrow \text{Term}) \rightarrow \text{Term}$. (We keep using the notation “ $\varsigma(x)b$ ” as syntactic sugar). Usual conventions about α -conversion apply.

2.1 Dynamic Semantics

The key point in using the Natural Deduction Semantics (NDS) [7, 23] is that all stack-like structures (*e.g.* environments) are distributed in the hypotheses of proof derivations. Therefore, judgments and proofs we have to deal with become appreciably simpler.

The *term reduction* judgment of $\text{imp}_{\varsigma}, \sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$, is translated as $\Gamma \vdash \text{eval}(s, a, s', v)$; that is, we model the operational semantics by a predicate *eval* defined on 4-tuples $\text{eval} \subseteq \text{Store} \times \text{Term} \times \text{Store} \times \text{Res}$. Γ is the *proof derivation context*, *i.e.* a set of assertions (of any judgment) which can be used as assumptions in the proof derivations. The intended meaning of the derivation $\Gamma \vdash \text{eval}(s, a, s', v)$ is that, starting with the store s and using the assumptions in Γ , the term a reduces to a result v , yielding an updated store s' . The rules for *eval* are in Figure 6: as usual in Natural Deduction, rules are written in “vertical” notation, *i.e.* the hypotheses of a derivation $\Gamma \vdash \mathcal{J}$ are distributed on the leaves of the proof tree.

Notice that the stack S disappears in the judgment *eval*: its content is distributed in Γ , *i.e.* Γ contains enough assumptions to carry the association between variables and results. These bindings are created in the form of hypothetical premises local to sub-reductions, discharged in the spirit of natural deduction style—see *e.g.* rules (*e.let*) and (*e.call*). It is worth noticing that we do not need to introduce the well-formedness

judgments for stores and stacks: these properties are automatically ensured by freshness conditions of *eigenvariables* in the natural deduction style.

A consequence of NDS is that closures cannot be pairs $\langle \text{method}, \text{stack} \rangle$, because there are no explicit stacks to put in anymore. Rather, we have to “calculate” closures by gathering from Γ the results associated to the free variables in the methods bodies. Closures *à la* Abadi and Cardelli $\langle \varsigma(x)b, S \rangle$ are translated as:

$$\lambda x. \text{bind}(v_1, \lambda y_1. \text{bind}(\dots \text{bind}(v_n, \lambda y_n. \text{ground}(b)) \dots))$$

where the first (outmost) abstraction λx stands for $\varsigma(x)$, and the n remaining abstractions ($n \geq 0$) capture the free variables of b . Hence, *bind* and *ground* are the two constructors of a new syntactic sort, *i.e.*: $\text{Body} : \bar{b} ::= \text{ground}(b) \mid \text{bind}(v, \lambda x. \bar{b})$.

Closures are dealt with by two auxiliary judgments $\text{wrap} \subseteq \text{Term} \times \text{Body}$ and $\text{eval}_b \subseteq \text{Store} \times \text{Body} \times \text{Store} \times \text{Res}$. The judgment *wrap* implements the formation of closure *bodies*, that is, terms where the only possibly free variable is the one corresponding to *self*. The intended meaning of $\Gamma \vdash \text{wrap}(b, \bar{b})$ is “ \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 the free variables in terms, we introduce a judgment $\text{closed} \subseteq \text{Term}$, whose formal meaning is $\Gamma \vdash \text{closed}(a) \iff \text{for all } x \in \text{FV}(a) : \text{closed}(x) \in \Gamma$. Intuitively, the rules of *wrap* allow for successively binding the free variables appearing in the method body (*w_bind*), until it is “closed” (*w_ground*). Notice that the closures we get in this way are “optimized”, because only variables which are really free in the body need to be bound in the closure.

Evaluation of closures takes place in the rule of method selection (*e_call*), in a context extended with the binding between a fresh variable (representing *self*) and the (implementation of) host object. Of course, all the local bindings of the closure have to be unraveled (*i.e.* assumed in the hypotheses) before the real evaluation of the body is performed; this unraveling is implemented by the auxiliary judgment eval_b , which can be seen as the dual of *wrap* and is defined by mutual induction with *eval*. For lack of space, we cannot describe in detail all the rules of Figure 6; we refer to [9] for a complete discussion.

Adequacy. We prove now on paper that the presentation of the operational semantics of imp_ς in NDS corresponds faithfully to the original one; see [9] for more details.

First, we establish the relationship between our heterogeneous context Γ and the environments S, E of the original setting, and between the two kinds of stores s and σ .

Definition 2 (Well-formed context). *A context Γ is well-formed if it can be partitioned as $\Gamma = \Gamma_{\text{Res}} \cup \Gamma_{\text{TType}} \cup \Gamma_{\text{closed}}$, where Γ_{Res} contains only formulae of the form $x \mapsto v$, and Γ_{TType} contains only formulae of the form $x \mapsto A$, and Γ_{closed} contains only formulae of the form $\text{closed}(x)$; moreover, Γ_{Res} and Γ_{TType} are functional (e.g., if $x \mapsto v, x \mapsto v' \in \Gamma_{\text{Res}}$, then $v \equiv v'$).*

Definition 3. *For Γ a context, S a stack, E a type environment, s, σ stores, we define:*

$$\begin{aligned} \Gamma \subseteq S &\triangleq \forall x \mapsto v \in \Gamma. x \mapsto v \in S & \Gamma \subseteq E &\triangleq \forall x:A \in \Gamma. x:A \in E \\ S \subseteq \Gamma &\triangleq \forall x \mapsto v \in S. x \mapsto v \in \Gamma & E \subseteq \Gamma &\triangleq \forall x:A \in E. x:A \in \Gamma \\ \gamma(S) &\triangleq \{x \mapsto S(x) \mid x \in \text{Dom}(S)\} \end{aligned}$$

$$\begin{array}{c}
\begin{array}{c}
(x \mapsto v) \\
\vdots \\
eval(s', b(x), s'', v') \\
eval(s, a, s', v)
\end{array}
\quad
\begin{array}{c}
(closed(x_i)) \\
\vdots \\
wrap(b_i, \bar{b}_i) \quad \iota_i \notin \text{Dom}(s) \\
s'' \equiv (s, \iota_i \mapsto \lambda x_i. \bar{b}_i)^{i \in I} \quad \forall i \in I
\end{array} \\
\hline
eval(s, let(a, b), s'', v') \quad (e_{let}) \qquad eval(s, [l_i = \varsigma(x_i) b_i]^{i \in I}, s'', [l_i = \iota_i]^{i \in I}) \quad (e_{obj})
\end{array}$$

$$\begin{array}{c}
(x \mapsto [l_i : \iota_i]^{i \in I}) \\
\vdots \\
eval_b(s', \bar{b}_j, s'', v) \quad j \in I \\
eval(s, a, s', [l_i : \iota_i]^{i \in I}) \quad s'(\iota_j) = \lambda x. \bar{b}_j
\end{array}
\quad
\begin{array}{c}
\iota_i, \iota'_i \notin \text{Dom}(s') \\
s'' \equiv s', \iota'_i \mapsto s'(\iota_i)^{i \in I} \\
eval(s, a, s', [l_i = \iota_i]^{i \in I}) \quad \forall i \in I
\end{array}
\quad
\begin{array}{c}
(closed(x_i)) \\
\vdots \\
closed(b_i) \quad \forall i \in I \\
closed([l_i = \varsigma(x_i) b_i]^{i \in I})
\end{array}$$

$$\begin{array}{c}
\begin{array}{c}
x \mapsto v \\
eval(s, x, s, v)
\end{array}
\quad
\begin{array}{c}
eval(s, a, s', v) \\
eval_b(s, ground(a), s', v)
\end{array}
\quad
\begin{array}{c}
(closed(x)) \\
\vdots \\
closed(b_i) \quad \forall i \in I \\
closed([l_i = \varsigma(x_i) b_i]^{i \in I})
\end{array}
\end{array}$$

$$\begin{array}{c}
\begin{array}{c}
(y \mapsto v) \\
\vdots \\
eval_b(s, \bar{b}, s', v')
\end{array}
\quad
\begin{array}{c}
(closed(x)) \\
\vdots \\
closed(a) \quad closed(b(x)) \\
closed(let(a, b))
\end{array}
\quad
\begin{array}{c}
closed(a) \\
closed(clone(a))
\end{array}
\end{array}$$

$$\begin{array}{c}
\begin{array}{c}
closed(a) \\
closed(a.l)
\end{array}
\quad
\begin{array}{c}
closed(b) \\
wrap(b, ground(b))
\end{array}
\quad
\begin{array}{c}
(closed(z)) \\
\vdots \\
wrap(b\{z/y\}, \bar{b}\{z/y\}) \quad z \text{ fresh} \\
wrap(b, bind(v, \lambda y. \bar{b}))
\end{array}
\end{array}$$

Fig. 6. Natural Deduction Dynamic Semantics for imp_{ς}^-

$s \lesssim \sigma \triangleq \forall \iota_i \in \text{Dom}(s). \gamma(S_i), closed(x_i) \vdash wrap(b_i, s(\iota_i)(x_i))$, and $\sigma(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle$
 $\sigma \lesssim s \triangleq \forall \iota_i \in \text{Dom}(\sigma). \gamma(S_i), closed(x_i) \vdash wrap(b_i, s(\iota_i)(x_i))$, and $\sigma(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle$

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

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

Theorem 2 (Adequacy of dynamic semantics). *Let Γ be well-formed, and $\sigma \cdot S \vdash \diamond$.*

1. *Let $\Gamma \subseteq S$, and $s \lesssim \sigma$.*
 - (a) *If $\Gamma \vdash eval(s, a, s', v)$, then there exists σ' such that $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$, and $s' \lesssim \sigma'$;*
 - (b) *If $\Gamma \vdash eval_b(s, \bar{b}, s', v)$, then there exists σ' such that $\sigma \cdot stack(\bar{b}) \vdash body(\bar{b}) \rightsquigarrow v \cdot \sigma'$, and $s' \lesssim \sigma'$.*

$$\begin{array}{c}
\frac{wt(B_i) \quad \forall i \in I}{wt([l_i : B_i]^{i \in I})} (wt_obj) \qquad \frac{wt(A)}{sub(A, A)} (sub_refl) \\
\\
\frac{sub(A, B) \quad sub(B, C)}{sub(A, C)} (sub_trans) \qquad \frac{wt(B_i) \quad \forall i \in I \cup J}{sub([l_i : B_i]^{i \in I \cup J}, [l_i : B_i]^{i \in I})} (sub_obj) \\
\\
\frac{type(a, A) \quad sub(A, B)}{type(a, B)} (t_sub) \qquad \frac{type(a, [l_i : B_i]^{i \in I}) \quad j \in I}{type(a.l_j, B_j)} (t_call) \\
\\
\frac{wt(A) \quad x \mapsto A}{type(x, A)} (t_var) \qquad \frac{type(a, [l_i : B_i]^{i \in I})}{type(clone(a), [l_i : B_i]^{i \in I})} (t_clone) \\
\\
\frac{\begin{array}{c} (x_i \mapsto [l_i : B_i]^{i \in I}) \\ \vdots \\ type(b_i, B_i) \quad \forall i \in I \end{array}}{type([l_i = \varsigma(x_i)b_i]^{i \in I}, [l_i : B_i]^{i \in I})} (t_obj) \qquad \frac{\begin{array}{c} (x \mapsto A) \\ \vdots \\ type(a, A) \quad type((b \ x), B) \end{array}}{type(let(a, b), B)} (t_let)
\end{array}$$

Fig. 7. Natural Deduction Static Semantics for imp_{ς}^-

2. Let $S \subseteq \Gamma$ and $\sigma \lesssim s$. If $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$, then there exists s' such that $\Gamma \vdash eval(s, a, s', v)$, and $\sigma' \lesssim s'$.

Proof. (1) By mutual induction on the structure of the derivations $\Gamma \vdash eval(s, a, s', v)$ and $\Gamma \vdash eval_b(s, \bar{b}, s', v)$. (2) By structural induction on $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$. \square

2.2 Static Semantics

The typing system for terms is easily rendered in NDS: the *term typing* judgment $E \vdash a : A$ is transformed as $\Gamma \vdash type(a, A)$, where $type \subseteq Term \times TType$. The *well-formedness* of types and the *subtyping* are also recovered in this setting, respectively as $wt \subseteq TType$ and $sub \subseteq TType \times TType$. As for stacks, well-formedness of the (distributed) typing environment is ensured by the freshness of locally quantified variables.

As the stack S disappears from the reduction judgment, so the type environment E disappears from the typing judgment, thus simplifying the judgment itself and the formal proofs about it: hence the global context Γ contains, among other assertions, the bindings between (free) variables and object types. The rules for term typing and related judgments in NDS are given in Figure 7. They consist in only a light transformation of the original ones: just notice that in the rules (t_obj) and (t_let) we discharge a typing assumption on a locally-quantified (*i.e.* fresh) variable, and that the premise $wt(A)$ in the rule (t_var) ensures that only well-formed types can be used for typing terms.

Theorem 3 (Adequacy of static semantics). *Let Γ be well-formed, and E such that $E \vdash \diamond$.*

$$\begin{array}{c}
\frac{
\begin{array}{c}
v \equiv [l_i = \iota_i]^{i \in I} \quad A \equiv [l_i : B_i]^{i \in I} \quad (x \mapsto A), (\text{cores}(s, v, A)) \\
s(\iota_i) \equiv \lambda x_i. \bar{b}_i \quad \text{wt}([l_i : B_i]^{i \in I}) \quad \text{cotype}_b(s, \bar{b}_i, B_i) \quad \forall i \in I \quad \iota_i \in \text{Dom}(s)
\end{array}
}{\text{cores}(s, v, A)} (t_{\text{cores}})
\\[10pt]
\frac{
\begin{array}{c}
\text{type}(b, A) \\
\text{cotype}_b(s, \text{ground}(b), A)
\end{array}
(t_{\text{coground}})
\quad
\frac{
\begin{array}{c}
(y \mapsto A) \\
\vdots \\
\text{cores}(s, v, A) \quad \text{cotype}_b(s, \bar{b}, B)
\end{array}
(t_{\text{cobind}})
}{\text{cotype}_b(s, \text{bind}(v, \lambda y. \bar{b}), B)}
}{\text{cotype}_b(s, \text{bind}(v, \lambda y. \bar{b}), B)}
\end{array}$$

Fig. 8. Natural Deduction Result Typing for imp_{Σ}^-

1. If $\Gamma \subseteq E$, and $\Gamma \vdash \text{type}(a, A)$, then $E \vdash a : A$;
2. If $E \subseteq \Gamma$, and $E \vdash a : A$, then $\Gamma \vdash \text{type}(a, A)$.

Proof. (1) By structural induction on the derivation of $\Gamma \vdash \text{type}(a, A)$.

(2) By structural induction on the derivation of $E \vdash a : A$. □

2.3 Result typing

In order to type results we need to type store locations, which, in turn, may contain other pointers to the store: thus, in this process potential loops may arise. A naïve system would chase pointers infinitely, unraveling non-wellfounded structures in the memory. The solution adopted in [1] (see Section 1) is to introduce yet another typing structure, *i.e.* *store types*, and further proof systems which assign to each location a type consistent with the content of the location. In proofs, store types have to be provided beforehand.

In this section, we propose here a different approach to result typing, using *coinduction* for dealing with non-wellfounded, circular data. This approach is quite successful for imp_{Σ}^- ; it is still an open question whether there exists an adequate coinductive semantics (without store types) for the full imp_{Σ} .

For imp_{Σ}^- , the two original judgments of *result typing* $\Sigma \models v : A$ and *store typing* $\Sigma \models \sigma$ collapse into a unique judgment $\Gamma \vdash \text{cores}(s, v, A)$. More precisely, we introduce a (potentially) coinductive predicate $\text{cores} \subseteq \text{Store} \times \text{Res} \times \text{TType}$, and an inductive one $\text{cotype}_b \subseteq \text{Store} \times \text{Body} \times \text{TType}$, which are mutually recursive defined. The intended meaning of the derivation of $\Gamma \vdash \text{cores}(s, v, A)$ is that the result v , containing pointers to the store s , has type A ; similarly, $\Gamma \vdash \text{cotype}_b(s, \bar{b}, A)$ means that the body \bar{b} in the store s has type A . The rules for these judgments are in Figure 8.

It is interesting to notice the way coinduction arises. The idea at the heart of the type system is simple and it can be caught by looking at rule (t_{cores}) . In order to check whether a result v can be given a type A , we open all the results (if any) belonging to the pointed method closures. Then, we visit and type recursively the store until we reach a closure without free variables, whose body can be typed using the traditional *type* judgment (t_{coground}) . If the original result v is encountered in this process, then its type is the type A we started with (t_{cobind}) , therefore the assertion we are proving has to be *assumed* in the hypotheses, hence the coinduction.

In conclusion, the typing system for results is very compact: we do not need store types (and all related machinery) anymore. And also, since stacks and type environments are already distributed in the proof contexts, we can dispense with the *stack typing* judgment. However, in stating and proving the Subject Reduction theorem, we will have to require that the types of the variables in the proof derivation context are consistent with the results associated to the same variables.

An example of coinductive proof is the following. We type the result $[l = 0]$ in the store (containing a loop) $s \triangleq \{0 \mapsto \lambda x. \text{bind}([l = 0], \lambda y. \text{ground}(y))\}$, or $\sigma \triangleq \{0 \mapsto \langle \varsigma(x)y, (y \mapsto [l = 0]) \rangle\}$ in the original notation of [1], as follows:

$$\begin{array}{c}
\frac{(y \mapsto [l : []])_{(2)}}{\text{type}(y, [l : []])} \text{ (t.var)} \quad \text{sub}([l : []], []) \\
\hline
\text{type}(y, []) \text{ (t.sub)} \\
\hline
\frac{(\text{cores}(s, [l = 0], [l : []]))_{(1)} \quad \text{cotype}_b(s, \text{ground}(y), []) \text{ (t.coground)}}{\text{cotype}_b(s, \text{bind}([l = 0], \lambda y. \text{ground}(y)), [])} \text{ (t.cobind) (2)} \\
\hline
\text{cores}(s, [l = 0], [l : []]) \text{ (t.cores) (1)}
\end{array}$$

Adequacy. Since we use coinductive proof systems, our perspective is quite different w.r.t. the original formulation of imp_ς . However, we have the following result.

Theorem 4 (Adequacy of result typing). *Let Γ be a well-formed context.*

1. *For $s \lesssim \sigma$, if $\Gamma \vdash \text{cores}(s, v, A)$, then there exists Σ such that $\Sigma \models v : A$, and $\Sigma \models \sigma$.*
2. *For $\sigma \lesssim s$, if $\Sigma \models v : A$, and $\Sigma \models \sigma$, then $\Gamma \vdash \text{cores}(s, v, A)$.*

Proof. (1) (2) By inspection on the hypothetical derivations. \square

2.4 Subject Reduction

Due to the new presentation, for stating and proving the Subject Reduction we have just to require the coherence between types and results associated to the variables in the proof derivation context Γ . That is, given the store $s: \forall x, w, C. x \mapsto w, x:C \in \Gamma \Rightarrow \Gamma \vdash \text{cores}(s, w, C)$. This hypothesis corresponds to the *stack typing* judgment of [1], but our management, thanks to distributed stacks and environments, is easier. Finally, we obtain the following version of the Subject Reduction theorem, which is considerably simpler both to state and prove w.r.t. the original one.

Theorem 5 (Subject Reduction for imp_ς^-). *Let Γ be a well-formed context. If $\Gamma \vdash \text{type}(a, A)$, and $\Gamma \vdash \text{eval}(s, a, t, v)$, and $\forall x, w, C. (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash \text{cores}(s, w, C)$, then there exists a type A^+ such that $\text{cores}(t, v, A^+)$, and $\text{sub}(A^+, A)$.*

Proof. By structural induction on the derivation $\Gamma \vdash \text{eval}(s, a, t, v)$; see [10]. \square

3 Formalization in $\text{CC}^{(\text{Co})\text{Ind}}$

The formalization of our novel presentation of imp_ς^- in the specification language of a proof assistant is still a complex task (although much simpler than formalizing the original system of [1]), because we have to face some subtle details which are left “implicit” on the paper. We can discuss here only some of the aspects of this development in $\text{CC}^{(\text{Co})\text{Ind}}$, the specification language of Coq; see [9, 11] for further details.

Syntax. A well-known problem we have to address is the treatment of the *binders*, namely ς and *let*. Binders are difficult to deal with: we would like the metalanguage takes care of all the burden of α -conversion, substitutions, variable scope and so on. In recent years, several approaches have been proposed for the formal reasoning on structures with binders, essentially differing on the expressive power of the underlying metalanguage; see *e.g.* [16, 24–26] for more discussion.

Among the many possibilities, we have chosen the *second-order abstract syntax*, called also “weak HOAS” [17, 24]. In this approach, binding operators are represented by constructors of higher-order type [13, 24]. The main difference with respect to the full HOAS is that abstractions range over unstructured (*i.e.* non inductive) sets of *abstract variables*. In this way, α -conversion is automatically provided by the metalanguage, while substitution of terms for variables is not. This fits perfectly the needs for the encoding of imp_ς , since the language is taken up-to α -equivalence, and there is no need of substitution in the semantics.

The signature of the weak HOAS-based encoding of the syntax is the following:

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

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. The constructor *var* is declared as a coercion, thus it may omitted in the following. An alternative definition of objects would use the lists of Coq library, but this choice does not allow for defining by recursion on terms some fundamental functions, essential for the rest of the formalization (such as, for example, the non-occurrence of variables “ \notin ”).

Weak HOAS has well-known encoding methodologies [16, 24, 26]; therefore, the adequacy of the encoding *w.r.t.* the NDS presentation follows from standard arguments.

3.1 Dynamic Semantics

Due to lack of space, we discuss here only a selection of the datatypes necessary to represent the entities and operations for their manipulation, as required by the semantics. Locations and method names can be faithfully represented by natural numbers; this permits to define results and stacks:

```

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

```

The environmental information of the stack is represented as a function associating a result to each (declared) variable. This map is never defined effectively: $(\text{stack } x) = v$ corresponds to $x \mapsto v$, which is discharged from the proof environment but never proved as a judgment. Correspondingly, assumptions about `stack` will be discharged in the rules in order to associate results to variables.

On the other hand, stores cannot be distributed in the proof environment: instead, they are lists of method closures, the i -th element of the list is the closure associated to ι_i . Closures are bodies abstracted with respect to the *self* variable, where bodies inhabit an inductive higher-order datatype:

```

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

```

Some functions are needed for handling lists, *e.g.* for merging lists of pairs into single lists, generating new results from objects and results, etc.; see [11] for the code.

Extra notions and judgments. For simplifying the representation of operational semantics and the proofs in `Coq`, we can formalize the predicate *closed*, which is used in the formation of closures, as a function:

```

Parameter dummy : Var->Prop.
Fixpoint closed [t:Term] : Prop := Cases t of
  (var x) =>(dummy x) | (obj ml) =>(closed_obj ml)
  | (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:Obj] : 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 $(\text{closed } a) : \text{Prop}$ 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 $(\text{closed } a)$ goal is easily dealt with using the tactics provided by `Coq`. In the same way, we define also the functions $\text{notin} : \text{Var} \rightarrow \text{Term} \rightarrow \text{Prop}$ and $\text{fresh} : \text{Var} \rightarrow \text{Varlist} \rightarrow \text{Prop}$, which capture the “freshness” of a variable in a term and *w.r.t.* a list of variables, respectively (see [11]).

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 $((z:Var) \sim (y=z) \rightarrow (\text{notin } y \ (b \ z)))$ ensures that b is a “good context” for y , *i.e.* y does not occur free in b . Thus, the replacement $b\{z/y\}$ in the rule (w_bind) can be rendered simply as the application $(b \ z)$.

Term reduction. The semantics of Figure 6 is formalized by two inductive judgments:

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

Most of their rules are encoded straightforwardly; only notice that the rules for variables and *let* illustrate how the proof derivation context is used to represent the stack:

```
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)
```

The formalization of the rule (e_obj) is a good example of the kind of machinery needed for manipulating objects, closures, stores and results:

```
e_obj : (s:Store) (ml:Obj) (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)))
```

The function `alloc` simply appends the new list of closures to the old store. The function `new_res_obj` produces a new result, collecting method names of the given object and pairing them with new pointers to the store. The function `scan` builds the closures using `wrap` and returns a predicate, whose validity ensures that the object methods have distinct names.

The method selection uses the extra predicate `eval_body` for evaluating closures:

```
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 evaluation of the body takes place in an environment where a local variable x denoting “self” is associated to (the value of) the receiver object itself. The predicate `eval_body` is defined straightforwardly.

3.2 Static Semantics

The encoding of the typing system for terms is not problematic. Like for stacks, we model the typing environment by means of a functional symbol, associating object types to variables:

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

Term typing is defined by mutual induction with the typing of objects; note that we need to carry along the whole (object) type while we scan and type the methods of objects:

```

Mutual Inductive type : Term->TType->Prop :=
| t_sub : (a:Term) (A,B:TType)
  (type a A)->(subtype A B)->(type a B)
| t_obj : (ml:Obj) (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)) (mk (nil (Lab*TType))))
| t_cons: (A,B,C:TType; ... m:Var->Term; pl:(list (Lab*TType)))
  (type_obj C (obj ml) A)->(wftype B)->(wftype C)->
  ((x:Var) (typenv x) = (C)->(type (m x) B))->
  ~(In l (labels A))->(list_from_type A) = (pl)->
  (type_obj C (obj (obj_cons l m ml)) (mk (cons (l,B) pl))).

```

where `subtype` represents the *sub* predicate. We omit here its encoding, which makes also use of an auxiliary predicate for permutation of lists representing object types.

3.3 Result Typing

The coinductive result typing system of Figure 8 is easily rendered in Coq by means of two mutual coinductive predicates. We only point out that, in the encoding of `cores`, we have to carry along the whole (result) type, as in the above typing of objects:

```

CoInductive cotype_body : Store->Body->TType->Prop :=
  t_ground: (s:Store) (b:Term) (A:TType)
    (type b A)->(cotype_body s (ground b) A)
| t_bind : (s:Store) (b:Var->Body) (A,B:TType) (v:Res)
  (cores A s v A)->
  ((x:Var)(typenv x) = (A)->(cotype_body s (b x) B))->
  (cotype_body s (bind v b) B)
with cores : TType->Store->Res->TType->Prop :=
  t_void : (A:TType) (s:Store)
    (cores A s (nil (Lab*Loc)) (mk (nil (Lab*TType))))
| t_step: (A,B,C:TType) (s:Store) (v:Res) (i:Loc) (c:Closure)
  (l:Lab) (pl:(list (Lab*TType)))
  (cores C s v A)->(store_nth i s) = (c)-> ...
  ((x:Var) (typenv x) = (C)->(cotype_body s (c x) B))->
  (cores C s (cons (l,i) v) (mk (cons (l,B) pl))).

```

4 Metatheory in Coq

One of the main aims of the formalization presented above is to allow for the formal development of important properties of $\text{imp}_{\mathcal{S}}$. In this section we discuss briefly the upmost important, yet delicate to prove, property of Subject Reduction. We have stated Subject Reduction for $\text{imp}_{\mathcal{S}}^-$ as Theorem 5, which is formalized in Coq as follows:

```

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

```


In order to prove the theorem, we have to address all the aspects concerning concrete structures, such as stores, objects, object types, results, and so on: thus, many technical lemmata about operational semantics, term and result typing have been formally proved. It turns out that these lemmata are relatively compact and easy to prove. In particular, we have carried out coinductive proofs for the `cores` predicate via the `Cofix` tactic; that is, we can construct infinitely regressive proofs by using the thesis as hypothesis, provided its application is guarded by introduction rules [15].

It is interesting to compare this development with that of the full `impΣ`, where store types and result typing are encoded inductively, close to the original setting [9, 10]. Due to the handling of store types, that encoding yields a formal development considerably more involved. Nevertheless, we can reuse with a minimal effort some (or part) of the proofs developed for the coinductive encoding, especially those not requiring an explicit inspection on the structure of store types. This re-usability of proofs enlightens the modularity of the present approach. At the same time, some properties concerning linear structures (such as stores) become much more involved when we have to manage also store types. In this case we cannot neither reuse, nor follow the pattern, of the proofs developed for `impΣ-`. This points out that managing bulky linear structures explicitly in judgments is unwieldy. We can conclude that the delegation of stacks and typing environments to the proof context, and the use of coinduction for dealing with store loops, reduce considerably the length and complexity of proofs.

Another key aspect of our formalization is the use of (weak) higher-order abstract syntax. Nowadays there is a lot of research towards a satisfactory support for programming with and reasoning about datatypes in higher-order abstract syntax, whose discussion is out of the scope of this paper. In the present case, the expressive power of `CC(Co)Ind` reveals to be not enough. A general methodology for adding the required extra expressive power, known as the *Theory of Contexts* (ToC), is presented in [17, 26]. The gist is to assume a small set of axioms capturing some basic and natural properties of (*variable*) *names* and *term contexts*. These axioms allow for a smooth handling of schemata in HOAS, with a very low mathematical and logical overhead. Thus this work can be seen also as an extensive case study about the application of the Theory of Contexts, used here for the first time on an imperative object-oriented calculus.

The Theory of Contexts allows to create “fresh” variables, via the *unsaturation axiom*: “ $\forall M. \exists x. x \notin M$ ”. This axiom has been introduced in an untyped setting. In our case, we have to take into account also the informations associated to the variables, such as results and types. More precisely, we adopt the unsaturation axiom in two flavours. The first one corresponds to the case of using metavariables as *placeholders*:

```
Axiom unsat : (A:TType; xl:Varlist)
              (EX x|(fresh x xl)/\ (dummy x)/\ (typenv x)=A).
```

The second axiom reflects the use of metavariables for *variables* of the object language; we assume the existence of fresh names to be associated both to results and their type, provided they are consistent in a given store. This corresponds exactly to the stack typing judgment in [1]:

```
Axiom unsat_cores : (s:Store) (v:Res) (A:TType) (cores s v A)->
                  (xl:Varlist)(EX x|(fresh x xl)/\ (stack x)=v/\ (typenv x)=A).
```

Both axioms can be validated in models similar to that of the original ToC.

5 Conclusions and future work

In this paper, we have studied the formal development of the theory of object-based calculi with types and side effects, such as imp_ζ , in type-theory based proof assistants, such as Coq . Before encoding the syntax and semantics of the calculus, we have taken advantage of the features offered by coinductive type theories, such as $\text{CC}^{(\text{Co})\text{Ind}}$. This perspective has suggested an original, and easier to deal with, presentation of the very same language, in the setting of *natural deduction semantics* (NDS) and *weak higher-order abstract syntax* (HOAS), and, for a significant fragment of imp_ζ , *coinductive* typing systems. This reformulation is interesting *per se*; moreover the absence of explicit linear structures (environments and store types) in the judgments has a direct impact on the structure of proofs, thus allowing for a simpler and smoother treatment of complex (meta)properties. The complete system has been encoded in Coq , and the fundamental property of Subject Reduction formally proved.

To our knowledge, this is the first development of the theory of an object-based language with side effects, in Logical Frameworks. The closest work may be [22], 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 [2].

Our experience leads us to affirm that the approach we have followed in the present work, using Coinductive Natural Deduction Semantics and HOAS with the Theory of Contexts, is particularly well-suited with respect to proof practice, also in the very challenging case of a calculus with objects, methods, cloning, dynamic lookup, types, subtypes, and imperative features. In particular, the use of coinduction seems to fit naturally the semantics of object calculi with side effects; therefore, the development of coinductive predicates and types in existing logical frameworks and proof environments should be pursued at a deeper extent.

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

Then, we plan to extend the development presented in this paper and in [10] to other object-based calculi, for instance those featuring *object extensions* [14], or *recursive types* [1]. The latter case could benefit again from coinductive types and predicates.

From a practical point of view, the formalization of imp_ζ can be used for the development of *certified* tools, such as interpreters, compilers and type-checking algorithms. Rather than *extracting* these tools from proofs, we plan to *certify* a given tool with respect to the formal semantics of the object calculus and the target machine. Some related results along this line regard the use of Coq and *Isabelle* for certifying compilers for an imperative language [5] and Java [27]. However, none of these works adopts higher-order abstract syntax for dealing with binders; we feel that the use of NDS and HOAS should simplify these advanced tasks in the case of languages with binders.

Acknowledgments. The authors are grateful to Yves Bertot, Joëlle Despeyroux, and Bernard Paul Serpette for fruitful discussions on earlier formalisations of imp_ζ .

References

1. M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.
2. M. Abadi and K. Leino. A logic of object-oriented programs. In *Proc. of TAPSOFT*, LNCS 1214. Springer-Verlag, 1997.
3. H. Barendregt and T. Nipkow, editors. *Proc. of TYPES*, LNCS 806. Springer-Verlag, 1994.
4. G. Barthe, P. Courtieu, G. Dufay, and S. M. de Sousa. Tool-assisted specification and verification of the JavaCard platform. In *Proc. of AMAST*, LNCS 2422, 2002.
5. Y. Bertot. A certified compiler for an imperative language. Technical Report INRIA, 1998.
6. Y. Bertot. Formalizing a JVMML verifier for initialization in a theorem prover. In *Proc. of CAV*, LNCS 2102. Springer-Verlag, 2001.
7. R. Burstall and F. Honsell. Operational semantics in a natural deduction setting. In *Logical Frameworks*. Cambridge University Press, 1990.
8. L. Cardelli. Obliq: A Language with Distributed Scope. *Computing Systems*, 1995.
9. A. Ciaffaglione. *Certified reasoning on Real Numbers and Objects in Co-inductive Type Theory*. PhD thesis, Dipartimento di Matematica e Informatica, Università di Udine, Italy and LORIA-INPL, Nancy, France, 2003.
10. A. Ciaffaglione, L. Liquori, and M. Miculan. On the formalization of imperative object-based calculi in (co)inductive type theories. Technical Report INRIA, 2003.
11. A. Ciaffaglione, L. Liquori, and M. Miculan. The Web Appendix of this paper, 2003. <http://www.dimi.uniud.it/~ciaffagl/Objects/Imp-covarsigma.tar.gz>.
12. J. Despeyroux. Proof of translation in natural semantics. In *Proc. of LICS 1986*. ACM, 1986.
13. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order syntax in Coq. In *Proc. of TLCA*, LNCS 905. Springer-Verlag, 1995.
14. K. Fisher, F. Honsell, and J. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1994.
15. E. Giménez. Codifying guarded recursion definitions with recursive schemes. In *Proc. of TYPES*, LNCS 996. Springer-Verlag, 1995.
16. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 1993.
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. Springer-Verlag, 2001.
18. M. Huisman. *Reasoning about Java programs in higher order logic with PVS and Isabelle*. PhD thesis, Katholieke Universiteit Nijmegen, 2001.
19. INRIA. *The Coq Proof Assistant*, 2003. <http://coq.inria.fr/doc/main.html>.
20. G. Kahn. Natural Semantics. In *Proc. of TACS*, LNCS 247. Springer-Verlag, 1987.
21. G. Klein and T. Nipkow. Verified bytecode verifiers. *TCS 298(3)*, 2003.
22. O. Laurent. Sémantique Naturelle et Coq : vers la spécification et les preuves sur les langages à objets. Technical Report INRIA, 1997.
23. M. Miculan. The expressive power of structural operational semantics with explicit assumptions. In [3].
24. M. Miculan. *Encoding Logical Theories of Programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1997.
25. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. of ACM SIGPLAN*, 1988.
26. I. Scagnetto. *Reasoning about Names In Higher-Order Abstract Syntax*. PhD thesis, Dipartimento di Matematica e Informatica, Università di Udine, 2002.
27. M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. of CADE*, LNCS 2392. Springer-Verlag, 2002.
28. H. Tews. A case study in coalgebraic specification: memory management in the FIASCO microkernel. Technical report, TU Dresden, 2000.
29. J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of JavaCard's application identifier class. In *Proc. of the JavaCard 2000 Workshop*, LNCS 2041, 2001.