Reasoning about Object-based Calculi in (Co)Inductive Type Theory and the Theory of Contexts

Alberto Ciaffaglione, Luigi Liquori, Marino Miculan

Reasoning about Object-based Calculi in (Co)Inductive Type Theory and the Theory of Contexts *

Alberto Ciaffaglione
Dipartimento di Matematica e Informatica, Università di Udine, Italy

Luigi Liquori
INRIA, Sophia Antipolis, France

Marino Miculan
Dipartimento di Matematica e Informatica, Università di Udine, Italy

Abstract. We illustrate a methodology for formalizing and reasoning about Abadi and Cardelli's object-based calculi, in (co)inductive type theory, such as the *Calculus of (Co)Inductive Constructions*, by taking advantage of *Natural Deduction Semantics* and *coinduction* in combination with *weak Higher-Order Abstract Syntax* and the *Theory of Contexts*.

Our methodology allows to implement smoothly the calculi in the target metalanguage; moreover, it suggests novel presentations of the calculi themselves. In detail, we present a compact formalization of the syntax and semantics for the functional and the imperative variants of the ς-calculus. Our approach simplifies the proof of Subject Reduction theorems, which are proved formally in the proof assistant Coq with a relatively small overhead.

Keywords: Functional and Imperative Object-calculi, Logical Foundations of Programming, Coinductive Type Theories, Logical Frameworks, Interactive Theorem Proving.

1. Introduction

In this paper, we present a methodology for formal reasoning about *object-based* calculi, aiming to take most advantage of the features offered by Logical Frameworks based on *coinductive type theories*. We illustrate this methodology by means of an extensive case study, about Abadi-Cardelli's ς-calculus (in both functional and imperative versions), using the *Calculus of (Co)Inductive Constructions* ($\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$) in its Coq implementation (Coq, 2003).

There are several motivations for this work. First, in recent years much effort has been put in formalizing *class-based* object-oriented languages (such as Java, C^{++} and $\mathrm{C}^{\#}$) in Coq, Isabelle and PVS (Marché et al., 2004; Klein and Nipkow, 2003; Huisman, 2001; Van den Berg et al., 2001; Tews, 2000). On the other hand, *object-based* languages, such as Self (Self, 2003) and Obliq (Cardelli, 1995) have received little attention. We see this fact as a serious gap, because most of the foundational calculi introduced for the mathematical analysis of the object-oriented paradigm are object-based (Abadi and Cardelli, 1996; Fisher et al., 1994). Indeed, object-based languages simplify and generalize class-based ones: they reduce classes to more primi-

* Supported by UE project IST-CA-510996 *Types* and French grant CNRS *ACI Modulogic*.

tive notions, provide more flexible mechanisms, and can be even used as intermediate code for the implementation of the latter.

Secondly, formalizing and reasoning about object-based calculi in a Logical Framework is challenging from the point of view of program certification. Often, object calculi summarize features usually found in different languages: objects, variable bindings, closures, functional and imperative method-update, stores, aliasing, circular pointers, types and subtyping, all at once. This level of complexity has a bearing in proving properties about the calculi: for instance, the property of Subject Reduction is much harder to state and prove for object-based languages than for pure functional ones. It is clear that this scenario can benefit from the use of Logical Frameworks: on one hand, the rigorous encoding in a metalanguage forces to spell out in full detail all the aspects of the calculus, thus giving the possibility to identify and fix problematic issues which are skipped on paper; on the other hand, the encoding methodology may offer the occasion for reformulating the calculus itself, which can be seen from novel, cleaner perspectives.

Now, a common problem is that encoding and reasoning about a formal system in a Logical Framework, adds further complexity to already cumbersome judgements and proofs. In order to be practically useful, therefore, it is important that the formalization is as clean and compact as possible. A typical example is the handling of bound variables: in spite of the fact that α-equivalence is taken for granted on paper, it does not hold *e.g.* in first-order encodings, where one has to deal explicitly with different representations of equivalent terms. Thus, an encoding of object-based calculi using traditional first-order techniques, although feasible, is not satisfactory, as it would yield a clumsy and unmanageable set of definitions, whose handling would add further difficulties to the formal development.

Therefore, a "good" encoding methodology should strive for simplicity: the overhead introduced by the formalization should be as low as possible. This prerequisite allows for simplifying the formal proof of complex metatheoretical results, such as Subject Reduction. Ideally, most (if not all) details implicitly taken for granted working with paper and pencil should be automatically provided in the formal development. A way for pursuing this goal is to *internalise* these issues *in* the metalanguage to the best extent, so that all the burden of their management is delegated to the Logical Framework. In the case of the ς-calculus, since the target metalanguage is $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, we have aimed to take most advantage of *hypothetic-general judgements*, *coinduction* and *weak higher-order abstract syntax* (HOAS).

The first issue is that the semantics of the ς-calculus is specified by means of several sequent-style systems: *à la* Kahn's *Natural Semantics*. Sequents contain explicit structures such as typing environments, evaluation stacks, stores, store types, etc. A straightforward representation of these structures as lists would lead to complicated judgements and proofs. Then, follow-

ing (Burstall and Honsell, 1990; Miculan, 1994), we use hypothetic-general judgements *à la* Martin-Löf for internalising those structures which obey to a stack-discipline. Hence, stacks and typing environments "disappear" from the formal judgements and proofs, which in turn become fairly simpler than the original ones. However, stack internalisation comes not for free: we have to provide a different management of closures. Far from being a problem, this suggests a novel formulation in *natural deduction* style of the ς-calculus, where closures are managed more efficiently than in the original version. On the other hand, stores are not stack-like structures, and hence cannot be internalised in an intuitionistic framework such as $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$. Nevertheless, we try to reduce their impact as much as possible.

A quite important consequence of having a store-based operational semantics is that the typing of values is not trivial, due to the potential presence of circular data structures ("pointer loops") in the store. The solution devised in (Abadi and Cardelli, 1996) is to use *store types*, which are auxiliary structures assigning a type to each location of a store compatibly with its content; however, these structures are not easy-to-use in a proof assistant. Luckily, nowadays type theories provide *coinduction* for dealing with circular, non well-founded entities (Giménez, 1995). Inspired by this feature, we elaborate an original coinductive system for typing values, without using store types, and instead recovering the types from the content of store locations. Using our system, whose expressive power is equivalent to the original one, we simplify further the proof of Subject Reduction for the functional version of the ς-calculus. (We cannot do the same for the imperative calculus, mainly because we cannot ensure to recover exactly the same type information along the computation, since the content of locations may change.)

Finally, when we come to the implementation in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, we have to face the problem of representing binders efficiently. To this end, one of the most suited approaches is *higher-order abstract syntax* (Pfenning and Elliott, 1988; Harper et al., 1993; Miculan, 1997). More precisely, since we work in a type theory with induction, we use *weak* HOAS (Miculan, 1997; Honsell et al., 2001b): binders are represented as *second*-order term constructors, taking as arguments functions over a parametric, open (*i.e.*, non inductive) type of *variables*. In this way, α-conversion of abstractions is automatically ensured by the parametricity of the set of variables, still retaining the benefits of inductive definitions and without the presence of *exotic terms* (Despeyroux et al., 1995). The main drawback of (weak) HOAS is that it is difficult to reason *about* the encodings. For instance, for proving Subject Reduction we have to prove several properties concerning *variable renaming*, often by induction over second-order terms. This is problematic, because $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ and similar type theories are not expressive enough (Honsell et al., 2001a). In order to overcome this problem, we adopt the *Theory of Contexts* (ToC) (Honsell et al., 2001a), a small set of axioms which can be added to the

existing logical framework ($CC^{(Co)Ind}$, in this case) to represent some basic and natural properties of variables and term contexts. These axioms have been proved to be consistent with (classical) higher order logic (Bucalo et al., 2006) (although their soundness in higher-order type theories is still under investigation). The main advantage of this approach is that it requires a very low mathematical and logical overhead: the arguments on paper can be readily ported to the formal setting, and it can be used in many existing proof environments without the need of any redesign of such systems.

To sum up, we present the first systematic formalization of (Abadi and Cardelli's) object-based calculi, in proof assistants based on type theories (the closest works are (Laurent, 1997; Gillard, 2000), which deal with functional semantics only, and (Hofmann and Tang, 2000), that does not formalize the operational semantics directly). We believe that the work described in this paper is an advancement both for the theory of object-based calculi and the pragmatics of interactive proof theory within the current generation of proof assistants. Their theoretical development and implementation will benefit from complex case studies such as the present one, where we test the applicability of advanced encoding and proof methodologies.

Synopsis. In Section 2 we recall the functional and imperative versions of the ς-calculus. In Section 3 we reformulate these calculi, bearing in mind the natural deduction approach and by taking advantage also of coinduction; the new formulations are proved to be equivalent to the original ones. The formalization in Coq of the new presentations, using weak Higher-Order Abstract Syntax, is discussed in Section 4. The formal development of meta-theoretic properties, such as the Subject Reduction theorem, using the Theory of Contexts, is presented in Section 5. Related work, conclusions and directions for future work are in Section 6. Longer proofs are reported in Appendix.

This paper is a revised and considerably extended version of two conference papers (Ciaffaglione et al., 2003a; Ciaffaglione et al., 2003b). The Coq code is available at (Ciaffaglione et al., 2003c).

2. The ς-Calculus

The ς-calculus is a calculus of objects, introduced by Abadi and Cardelli as the kernel of the languages Obliq (Cardelli, 1995) and Self (Self, 2003). We focus here on its (untyped) *functional* and *imperative* variants, both equipped with first-order typing *à la* Curry (Abadi and Cardelli, 1996, Ch. 6, 10, 11). We first describe the functional $\mathsf{fun}\varsigma$, then extend it to the imperative $\mathsf{imp}\varsigma$.

2.1. THE FUNς-CALCULUS

Syntax. The syntax of terms is given in Figure 1. Variables are taken from an infinite set $Var = \{x_1, x_2, \dots\}$ of distinct symbols, ranged over by x, y, z.

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

Figure 1. Syntax of the functional calculus $\mathsf{fun}\varsigma$.

We give here a brief explanation of the intuitive meaning of the constructs.

An *object* is a collection of *components* $l_i{=}\varsigma(x_i)b_i$, $i \in I$, with distinct method names l_i and associated methods $\varsigma(x_i)b_i$. The order of the components does not matter; the locally bound variable x_i (also called "self") denotes the *host* object, that is, the object containing the methods $\{l_i\}_{i\in I}$.

Method invocation $a.l$, where the method named l in a is $\varsigma(x)b$, has the intent of executing the method-body b with the parameter x bound to the host object a, then returning the result of the execution.

Method update $a.l{\leftarrow}\varsigma(x)b$ has here a *functional* meaning: a *new* object is built via the up to date methods of the old object and the new method $\varsigma(x)b$.

In all cases, ς acts as a binder: *e.g.*, in $\varsigma(x).b$, x is bound in b. Usual conventions about α-conversion and free variables (denoted by $\mathsf{FV}(a)$) apply.

Dynamic Semantics. The operational semantics of $\mathsf{fun}\varsigma$ is expressed by a big-step reduction relation (Kahn, 1987; Despeyroux, 1986), relating two stores σ, σ', a stack S, a term a, and a result v:

$$\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow v{\cdot}\sigma'$$

The intended meaning is that, starting with the store σ (playing the role of a *heap*) 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. More precisely, the entities involved in the semantics belong to the following sorts:

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

In this semantics, variables are never replaced by terms: they are associated to values, *i.e. (object) results*, by *stacks*. A result represents an object: it is a collection of method labels together with the locations where the corresponding *(method) closures* are stored. Closures are pairs built by a *method* $\varsigma(x_i)b_i$ and a stack S_i, such that $\mathsf{FV}(\varsigma(x_i)b_i) \subseteq \mathrm{dom}(S_i)$. Finally, locations are associated to closures by *stores*, which are (finite) functions. Unless differently remarked, all the l_i, ι_i, x_i are distinct.

This semantics differs from the original one in (Abadi and Cardelli, 1996, Chapter 6), where a *substitution-based* semantics (*i.e.* without stacks and

$$\frac{}{\emptyset \vdash_{\mathsf{AC}} \Diamond}(\mathrm{Store}\cdot\emptyset) \qquad \frac{\sigma\cdot S \vdash_{\mathsf{AC}} \Diamond \quad \iota \notin \mathsf{Dom}(\sigma)}{\sigma, \iota{\mapsto}\langle \varsigma(x)b, S\rangle \vdash_{\mathsf{AC}} \Diamond}(\mathrm{Store}\cdot\iota) \qquad \frac{\sigma \vdash_{\mathsf{AC}} \Diamond}{\sigma\cdot\emptyset \vdash_{\mathsf{AC}} \Diamond}(\mathrm{Stack}\cdot\emptyset)$$

$$\frac{\sigma\cdot S \vdash_{\mathsf{AC}} \Diamond \quad x \notin \mathsf{Dom}(S) \quad \forall i \in I : \iota_i \in \mathsf{Dom}(\sigma)}{\sigma\cdot(S, x{\mapsto}[l_i = \iota_i]^{i\in I}) \vdash_{\mathsf{AC}} \Diamond}(\mathrm{Stack}\cdot\mathrm{Var})$$

Figure 2. Well-formedness for Store and Stack.

stores) is given. We consider the finer-grained semantics presented in this paper for several reasons. First, we do not need to define (and reason about) any machinery for implementing substitution. Secondly, the given semantics is closer to actual implementation techniques on register-based machines, making explicit how stacks and stores are implemented; in this way, we can reason at a deeper level of detail, exposing to the certification process also aspects which would be swept under the carpet if we adopted a purely functional approach. Third, it will be easier to extend the semantics to the imperative features of $\mathsf{imp}\varsigma$, later on. Nevertheless, it is easy to see that our presentation is equivalent to the original one (just erasing the extra structures).

In the following, $\iota_i{\mapsto}\langle \varsigma(x)b, S\rangle_i^{\,i\in I}$ represents the store that maps the locations ι_i to the closures $\langle \varsigma(x)b, S\rangle_i$, for $i \in I$, and $\sigma, \iota{\mapsto}\langle \varsigma(x)b, S\rangle$ denotes the store σ extended with $\langle \varsigma(x)b, S\rangle$ at location ι (fresh), and $\sigma.\iota{\leftarrow}\langle \varsigma(x)b, S\rangle$ replaces the content of the location ι of σ with $\langle \varsigma(x)b, S\rangle$.

Stores and stacks are subject to well-formedness conditions, which are represented by two auxiliary judgements, $\sigma \vdash_{\mathsf{AC}} \Diamond$ and $\sigma\cdot S \vdash_{\mathsf{AC}} \Diamond$ (Figure 2). The rules for the reduction judgement are given in Figure 3. In particular, the *functional* method update $(\mathrm{Red}\cdot\mathrm{Upd_F})$ allocates a fresh location for storing the new method. (Thus the old location may become garbage, if there are no other references to it, but in this paper we do not address garbage collection.) Notice that an algorithm for reduction can be easily extracted from the rules.

Static Semantics. $\mathsf{fun}\varsigma$ is equipped with a first-order typing system with subtyping. The only type constructor is the one for object types, *i.e.*:

$$\mathit{TType}: \ A, B ::= [l_i{:}A_i]^{i\in I} \quad (l_i \text{ distinct})$$

so the only ground type is $[\,]$, which can be used for building object types; other ground types, as *e.g.* `bool`, `nat`, `int`, `real`, can be added at will.

The type system is given by four judgements: well-formedness of the type environment $E \vdash_{\mathsf{AC}} \Diamond$, well-formedness of object types $E \vdash_{\mathsf{AC}} A$, subtyping $E \vdash_{\mathsf{AC}} A <: B$, and term typing $E \vdash_{\mathsf{AC}} a : A$, where the typing environment E consists of assignments of (object) types to variables, each of the form $x{:}A$. The rules for all the judgements are collected in Figures 4 and 5.

The subtyping relation induces the notion of *subsumption*: an object of a given type also belongs to any supertype of that type and can subsume

$$\frac{\sigma{\cdot}(S', x{\mapsto}v, S'') \vdash_{\mathsf{AC}} \Diamond}{\sigma{\cdot}(S', x{\mapsto}v, S'') \vdash_{\mathsf{AC}} x \rightsquigarrow v{\cdot}\sigma}(\text{Red}{\cdot}\text{Var})$$

$$\frac{\sigma{\cdot}S \vdash_{\mathsf{AC}} \Diamond \quad \forall i \in I : \iota_i \notin \mathsf{Dom}(\sigma)}{\sigma{\cdot}S \vdash_{\mathsf{AC}} [l_i = \varsigma(x_i)b_i]^{i\in I} \rightsquigarrow [l_i = \iota_i]^{i\in I}{\cdot}(\sigma, \iota_i{\mapsto}\langle\varsigma(x_i)b_i, S\rangle^{i\in I})}(\text{Red}{\cdot}\text{Obj})$$

$$\frac{\begin{array}{ll}\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow [l_i = \iota_i]^{i\in I}{\cdot}\sigma' \quad j \in I & \sigma'(\iota_j) = \langle\varsigma(x_j)b_j, S'\rangle \\ x_j \notin \mathsf{Dom}(S') & \sigma'{\cdot}(S', x_j{\mapsto}[l_i = \iota_i]^{i\in I}) \vdash_{\mathsf{AC}} b_j \rightsquigarrow v{\cdot}\sigma''\end{array}}{\sigma{\cdot}S \vdash_{\mathsf{AC}} a.l_j \rightsquigarrow v{\cdot}\sigma''}(\text{Red}{\cdot}\text{Sel})$$

$$\frac{\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow [l_i = \iota_i]^{i\in I}{\cdot}\sigma' \quad \iota'_j \notin \mathsf{Dom}(\sigma') \quad j \in I}{\sigma{\cdot}S \vdash_{\mathsf{AC}} a.l_j{\leftarrow}\varsigma(x)b \rightsquigarrow [l_i = \iota_i, l_j = \iota'_j]^{i\in I\setminus\{j\}}{\cdot}(\sigma', \iota'_j{\mapsto}\langle\varsigma(x)b, S\rangle)}(\text{Red}{\cdot}\text{Upd}_{\text{F}})$$

Figure 3. Natural Operational Semantics for $\mathsf{fun}\varsigma$.

$$\frac{}{\emptyset \vdash_{\mathsf{AC}} \Diamond}(\text{Env}{\cdot}\emptyset) \qquad \frac{E \vdash_{\mathsf{AC}} A \quad x \notin \mathsf{Dom}(E)}{E, x{:}A \vdash_{\mathsf{AC}} \Diamond}(\text{Env}{\cdot}\text{Var})$$

$$\frac{E \vdash_{\mathsf{AC}} \Diamond \quad \forall i \in I : \quad E \vdash_{\mathsf{AC}} A_i \quad l_i \text{ distinct}}{E \vdash_{\mathsf{AC}} [l_i{:}A_i]^{i\in I}}(\text{Type}{\cdot}\text{Obj})$$

$$\frac{E \vdash_{\mathsf{AC}} A <: B \quad E \vdash_{\mathsf{AC}} B <: C}{E \vdash_{\mathsf{AC}} A <: C}(\text{Sub}{\cdot}\text{Trans}) \qquad \frac{E \vdash_{\mathsf{AC}} A}{E \vdash_{\mathsf{AC}} A <: A}(\text{Sub}{\cdot}\text{Refl})$$

$$\frac{\forall i \in I \cup J : \quad E \vdash_{\mathsf{AC}} A_i \quad l_i \text{ distinct}}{E \vdash_{\mathsf{AC}} [l_i{:}A_i]^{i\in I\cup J} <: [l_i{:}A_i]^{i\in I}}(\text{Sub}{\cdot}\text{Obj})$$

Figure 4. Auxiliary Typing judgements.

objects in the supertype, 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: $[l_i{:}A_i]^{i\in I\cup J} <: [l_i{:}B_i]^{i\in I}$ requires $A_i \equiv B_i$ for all $i \in I$, *i.e.* shared labels have *invariant* (*i.e.* neither covariant nor contravariant) associated types. This condition guarantees the soundness of the type discipline.

2.2. THE IMPς-CALCULUS

The imperative calculus $\mathsf{imp}\varsigma$ extends $\mathsf{fun}\varsigma$ with *object cloning* and *side effects*. The syntax of $\mathsf{imp}\varsigma$ simply extends that of $\mathsf{fun}\varsigma$ (Figure 1) with the constructs in Figure 6, where let binds x in b.

The *cloning* operation builds a new object with the same labels and methods of a. The *let* construct evaluates the term a, binds the result to a variable x, and then evaluates b with the variable x in the scope. This allows to have local definitions and control the execution flow: for instance, sequential evaluation can be defined as $a; b \triangleq let\ x = a\ in\ b$, where $x \notin \mathsf{FV}(b)$.

$$\frac{E \vdash_{\mathsf{AC}} a : A \quad E \vdash_{\mathsf{AC}} A <: B}{E \vdash_{\mathsf{AC}} a : B}(\text{Val·Sub}) \qquad \frac{E', x{:}A, E'' \vdash_{\mathsf{AC}} \Diamond}{E', x{:}A, E'' \vdash_{\mathsf{AC}} x : A}(\text{Val·Var})$$

$$\frac{\forall i \in I : \quad E, x_i{:}C \vdash_{\mathsf{AC}} b_i : A_i}{E \vdash_{\mathsf{AC}} [l_i = \varsigma(x_i)b_i]^{i\in I} : C}(\text{Val·Obj}) \qquad \frac{E \vdash_{\mathsf{AC}} a : [l_i{:}A_i]^{i\in I} \quad j \in I}{E \vdash_{\mathsf{AC}} a.l_j : A_j}(\text{Val·Sel})$$

$$\frac{E \vdash_{\mathsf{AC}} a : C \quad E, x{:}C \vdash_{\mathsf{AC}} b : A_j \quad j \in I}{E \vdash_{\mathsf{AC}} a.l_j \leftarrow \varsigma(x)b : C}(\text{Val·Upd}) \qquad \text{where } C \equiv [l_i{:}A_i]^{i\in I}$$

Figure 5. Type System for $\mathsf{fun}\varsigma$.

$Term:\ a, b\ ::=$	$\ldots$	as in $\mathsf{fun}\varsigma$
	$clone(a)$	cloning
	$let\ x = a\ in\ b$	local declaration

Figure 6. Syntax of the imperative calculus $\mathsf{imp}\varsigma$.

The operational semantics is properly a modification and extension of that of $\mathsf{fun}\varsigma$, see Figure 7. Notice that now the method update is an *imperative* operation: it *replaces* the closure stored in the location pointed to by ι_j with the new closure, *without* allocating a new location, thus returning a modified object. Using this kind of update, it is possible to create *pointer loops*, *i.e.* circular references among locations in the store.

EXAMPLE 1. Let us consider the following evaluation:

$$\emptyset{\cdot}\emptyset \vdash_{\mathsf{AC}} [l = \varsigma(x)x.l \leftarrow \varsigma(y)x].l \rightsquigarrow [l{=}0]{\cdot}\sigma$$

Then, the store $\sigma \equiv 0 \mapsto \langle \varsigma(y)x,\ (x{\mapsto}[l{=}0]) \rangle$ *"contains a loop, because it maps the index* 0 *to a closure that binds the variable* x *to a value that contains index* 0*. Hence an attempt to read out the result of* $[l = \varsigma(x)x.l \leftarrow \varsigma(y)x].l$ *by "inlining" the store and stack mappings would produce the infinite term* $[l = \varsigma(y)[l = \varsigma(y)...]]$" (Abadi and Cardelli, 1996, pp 138-139). □

Finally, the type system for $\mathsf{imp}\varsigma$ extends that of $\mathsf{fun}\varsigma$ with the following two rules for the new constructs:

$$\frac{E \vdash_{\mathsf{AC}} a : C}{E \vdash_{\mathsf{AC}} clone(a) : C}(\text{Val·Clone}) \qquad \frac{E \vdash_{\mathsf{AC}} a : A \quad E, x{:}A \vdash_{\mathsf{AC}} b : B}{E \vdash_{\mathsf{AC}} let\ x = a\ in\ b : B}(\text{Val·Let})$$

2.3. TYPE SOUNDNESS

Type Soundness is a fundamental property of any typed calculus, ensuring that "well-typed programs cannot go wrong". In the present case, this means

$$(\mathrm{Red}\cdot\mathrm{Var}),\ (\mathrm{Red}\cdot\mathrm{Obj}),\ (\mathrm{Red}\cdot\mathrm{Sel})\text{: as in } \mathsf{fun}_\varsigma$$

$$\frac{\sigma\cdot S \vdash_{\mathsf{AC}} a \rightsquigarrow v'\cdot\sigma' \quad \sigma'\cdot(S, x{\mapsto}v') \vdash_{\mathsf{AC}} b \rightsquigarrow v''\cdot\sigma''}{\sigma\cdot S \vdash_{\mathsf{AC}} let\ x = a\ in\ b \rightsquigarrow v''\cdot\sigma''}(\mathrm{Red}\cdot\mathrm{Let})$$

$$\frac{\sigma\cdot S \vdash_{\mathsf{AC}} a \rightsquigarrow [l_i = \iota_i]^{i\in I}\cdot\sigma' \quad \forall i \in I : \iota'_i \notin \mathsf{Dom}(\sigma')}{\sigma\cdot S \vdash_{\mathsf{AC}} clone(a) \rightsquigarrow [l_i = \iota'_i]^{i\in I}\cdot(\sigma', \iota'_i{\mapsto}\sigma'(\iota_i)^{i\in I})}(\mathrm{Red}\cdot\mathrm{Clo})$$

$$\frac{\sigma\cdot S \vdash_{\mathsf{AC}} a \rightsquigarrow [l_i = \iota_i]^{i\in I}\cdot\sigma' \quad j \in I}{\sigma\cdot S \vdash_{\mathsf{AC}} a.l_j{\leftarrow}\varsigma(x)b \rightsquigarrow [l_i = \iota_i]^{i\in I}\cdot(\sigma'.\iota_j{\leftarrow}\langle\varsigma(x)b, S\rangle)}(\mathrm{Red}\cdot\mathrm{Upd_I})$$

Figure 7. Natural Operational Semantics for imp_ς.

that the evaluation of any well-typed and not diverging term of fun_ς and imp_ς will never invoke an undefined method (*i.e.* the runtime exception *message-not-found* is never raised). Type Soundness is an immediate consequence of the *Subject Reduction* theorem, which relates the dynamic semantics to the static semantics, stating that the result produced by the evaluation of a term can be given a type consistent with that of the term itself.

In order to state formally Subject Reduction, Abadi and Cardelli introduce a typing system for results (Abadi and Cardelli, 1996, Chapter 11); notice that such a typing system applies to both fun_ς and imp_ς, since the two calculi have the same notion of result. We recall that a result is essentially a list of pointers to store locations (on a par with method labels); thus, in order to type a result, it is necessary to type the contents of the locations it points to. Now, a location containing a method $\varsigma(x).b$ can be given a *method type* M, which is a type of the form $[l_i{:}B_i]^{i\in I} \Rightarrow B_j$ (where $j \in I$); here, $[l_i{:}B_i]^{i\in I}$ is intended to be the type of the bound variable x, and B_j the type of the j-th method body. Hence, a store can be given a *store type* Σ, which is a finite map $\Sigma ::= (\iota_i{\mapsto}M_i)^{i\in I}$, assigning a method type to each location.

The typing system for results and stores is composed by five judgements: well-formedness of method types $M \models \Diamond$ and store types $\Sigma \models \Diamond$, result typing $\Sigma \models v : A$, store typing $\Sigma \models \sigma$, and stack typing (*i.e.* compatibility) $\Sigma \models S : E$. The intended meaning of the main judgement $\Sigma \models v : A$ is that the result v is given the type A, using the types assigned to locations by Σ. More formally, using the projection functions $\pi_i(a_1, \ldots, a_n) = a_i$ $(i \leq n)$:

$$\Sigma \models v : A \iff \forall \iota_i = \pi_2(\pi_i(v)) : \Sigma_1(\iota_i) = A \ \wedge\ \Sigma_2(\iota_i) = \pi_2(\pi_i(A))$$

where Σ_1, Σ_2 are the "first" and "second" projections of store types:

$$\begin{array}{lll} \Sigma_1(\iota) & \triangleq & [l_i{:}B_i]^{i\in I} \quad \text{if } \Sigma(\iota) = [l_i{:}B_i]^{i\in I} \Rightarrow B_j \\ \Sigma_2(\iota) & \triangleq & B_j \qquad\qquad \text{if } \Sigma(\iota) = [l_i{:}B_i]^{i\in I} \Rightarrow B_j \end{array}$$

On the other hand, store typing $\Sigma \models \sigma$ ensures that the content of every store location in σ can be given the type assigned to the same location by Σ. The rules for all these judgements are collected in Figure 8.

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

$$\frac{\Sigma \models \Diamond \quad \forall i \in I \quad \Sigma_1(\iota_i) \equiv [l_i{:}\Sigma_2(\iota_i)]^{i\in I}}{\Sigma \models [l_i = \iota_i]^{i\in I} : [l_i{:}\Sigma_2(\iota_i)]^{i\in I}}(\text{Res}) \qquad \frac{\Sigma \models S_i : E_i \quad \forall i \in I \quad E_i, x_i{:}\Sigma_1(\iota_i) \vdash_{\mathsf{AC}} b_i : \Sigma_2(\iota_i)}{\Sigma \models \iota_i \mapsto \langle \varsigma(x_i) b_i, S_i \rangle^{i\in I}}(\text{Sto·Typ})$$

$$\frac{\Sigma \models \Diamond}{\Sigma \models \emptyset : \emptyset}(\text{Stk·}\emptyset) \qquad \frac{x \notin \mathsf{Dom}(S) \cup \mathsf{Dom}(E) \quad \Sigma \models S : E \quad \Sigma \models v : A}{\Sigma \models (S, x \mapsto v) : (E, x{:}A)}(\text{Stk·Var})$$

Figure 8. Typing system for results and stores.

It is important to point out that store types have been introduced for typing results in presence of loops in the store (see Example 1). Due to loops, it is not always possible to determine the type of a result by examining its substructures recursively, that is, by recursively chasing pointers starting from store locations pointed to by the original result (unless by using a *coinductive* typing system, as in Section 3.4).

Another aspect of fun_ς and imp_ς is that the *minimum type* property (*i.e.*, if a term a is typable, then it has a type τ such that for any type σ of a, it is $\tau <: \sigma$) does not hold[1]. store types allow to overcome the issue of the ambiguity of typing, by fixing a given "reference" type for a store.

Finally, observe that type-sound computations must store in a location only closures compatible with the type given by the store type; and notice that store types can be extended, but not overwritten.

DEFINITION 2 (store type extension). *Σ' is an* extension *of Σ ($\Sigma' \geq \Sigma$) if and only if $\mathsf{Dom}(\Sigma) \subseteq \mathsf{Dom}(\Sigma')$, and for all $\iota \in \mathsf{Dom}(\Sigma)$: $\Sigma'(\iota) = \Sigma(\iota)$.* □

The following Subject Reduction Theorem holds for both fun_ς and imp_ς.

THEOREM 3 (Subject Reduction). *If $E \vdash_{\mathsf{AC}} a : A$, and $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow v{\cdot}\sigma^\dagger$, and $\Sigma \models \sigma$, and $\mathsf{Dom}(\sigma) = \mathsf{Dom}(\Sigma)$, and $\Sigma \models S : E$, then there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$, such that $\Sigma^\dagger \geq \Sigma$, and $\Sigma^\dagger \models \sigma^\dagger$, and $\mathsf{Dom}(\sigma^\dagger) = \mathsf{Dom}(\Sigma^\dagger)$, and $\Sigma^\dagger \models v : A^\dagger$, and $A^\dagger <: A$.* □

See (Abadi and Cardelli, 1996, Chapter 11) for the complete proof in the case of imp_ς; the proof for fun_ς is similar (and simpler).

COROLLARY 4 (Type Soundness). *The reduction of a non-diverging well-typed term of fun_ς and imp_ς in a well-typed store cannot get stuck, and produces a result of the expected type.* □

[1] This property would hold if the calculus was extended with Church-style type annotations for bound ("self") variables in methods.

3. fun_{ς} and imp_{ς} in Natural Deduction Semantics

In this section we give an alternative presentation of fun_{ς} and imp_{ς}, inspired by the features of Logical Frameworks based on Type Theory. Following (Burstall and Honsell, 1990; Miculan, 1994), the various proof systems are reformulated in *natural deduction* style,[2] where all stack-like structures are distributed in the hypotheses of proof derivations, thus making judgements and proofs fairly simpler (at the expense of introducing some auxiliary judgements). We refer to this setting as *Natural Deduction Semantics* (NDS). Moreover, we present an alternative, *coinductive* typing system for results, a further refinement which allows to avoid the use of store types.

As usual in Natural Deduction, proof systems will be written in "vertical" notation: the hypotheses of a derivation $\Gamma \vdash_{\mathsf{ND}} \mathcal{J}$ are distributed on the leaves of the proof tree. (To save space, in the text we keep writing natural deduction judgements in "horizontal", sequent form.)

Syntax. In this section, we use the same syntax of the original presentation (Section 2), with one important difference: we do not enforce at the syntactic level that the labels of an object or a type are all different. The advantage of this choice is that the correspondence between these syntactic categories and their corresponding implementation in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ will be simplified. It is important to notice that the extra, ill-formed terms are harmless, because the well-formedness condition is enforced explicitly in the rules of the static and dynamic semantics (rules (e·obj) and (wt·obj)). For instance, the term $[l{=}\varsigma(x_1)b_1,\ l{=}\varsigma(x_2)b_2]$ is syntactically correct, but it cannot be typed nor evaluated, because typing and evaluation rules force all method labels of an object to be different.

3.1. Dynamic Semantics

The judgement $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow v{\cdot}\sigma'$ is translated as $\Gamma \vdash_{\mathsf{ND}} eval(s, a, s', v)$, where Γ denotes the *proof derivation context* (*i.e.* a set of assertions, of any judgement, which can be used as assumptions in the proof derivations), and $eval$ is a predicate defined on 4-tuples $eval \subseteq Store \times Term \times Store \times Res$.

The rules for $eval$ for fun_{ς} and imp_{ς} are in Figure 9 and 10, respectively.

The intended meaning of $\Gamma \vdash_{\mathsf{ND}} eval(s, a, s', v)$ is that, starting with the store s and using the assumptions in Γ, the term a reduces to a result v, providing an updated store s'. The content of a stack S, *i.e.* the associations between variables and results, is represented by suitable assumptions of the form "$x{\mapsto}v$" in the proof context Γ. These associations are created as hypothetical premises local to sub-reductions, and are discharged in the

[2] For our concerns, a "good" natural deduction system has essentially the same structural rules of intuitionistic logic, that is weakening, contraction and permutation.

$$\frac{x{\mapsto}v}{eval(s,x,s,v)}(\text{e·var})$$

$$\frac{\begin{array}{c}(closed(x_i))\\ \vdots\\ \forall i\in I:\iota_i\notin \mathsf{Dom}(s)\quad wrap(b_i,\overline{b}_i)\quad \forall i,j\in I, i\neq j: l_i\neq l_j\end{array}}{eval(s,[l_i=\varsigma(x_i)b_i]^{i\in I},(s,\iota_i{\mapsto}\lambda x_i.\overline{b}_i)^{i\in I},[l_i=\iota_i]^{i\in I})}(\text{e·obj})$$

$$\frac{\begin{array}{c}(x{\mapsto}[l_i=\iota_i]^{i\in I})\\ \vdots\\ eval(s,a,s',[l_i=\iota_i]^{i\in I})\quad s'(\iota_j)=\lambda x.\overline{b}_j\quad eval_b(s',\overline{b}_j,s'',v)\quad (j\in I)\end{array}}{eval(s,a.l_j,s'',v)}(\text{e·call})$$

$$\frac{\begin{array}{c}(closed(x))\\ \vdots\\ eval(s,a,s',[l_i=\iota_i]^{i\in I})\quad wrap(b,\overline{b})\quad \iota'_j\notin \mathsf{Dom}(s')\quad (j\in I)\end{array}}{eval(s,a.l{\leftarrow}\varsigma(x)b,(s',\iota'_j{\mapsto}\lambda x.\overline{b}),[l_i=\iota_i,l_j=\iota'_j]^{i\in I\setminus\{j\}})}(\text{e·upd}_\text{f})$$

$$\frac{\begin{array}{c}(y{\mapsto}v)\\ \vdots\\ eval_b(s,\overline{b},s',v')\end{array}}{eval_b(s,\overline{b}[y\mapsto v],s',v')}(\text{e·bind})$$

$$\frac{eval(s,a,s',v)}{eval_b(s,a,s',v)}(\text{e·ground})$$

Figure 9. Natural Deduction Dynamic Semantics for fun_ς.

$$\frac{\begin{array}{c}(closed(x))\\ \vdots\\ eval(s,a,s',[l_i=\iota_i]^{i\in I})\quad wrap(b,\overline{b})\quad (j\in I)\end{array}}{eval(s,a.l_j{\leftarrow}\varsigma(x)b,(s'.\iota_j{\leftarrow}\lambda x.\overline{b}),[l_i=\iota_i]^{i\in I})}(\text{e·upd}_\text{i})$$

$$\frac{eval(s,a,s',[l_i=\iota_i]^{i\in I})\quad \forall i\in I:\iota'_i\notin \mathsf{Dom}(s')}{eval(s,clone(a),(s',\iota'_i{\mapsto}s'(\iota_i)^{i\in I}),[l_i=\iota'_i]^{i\in I})}(\text{e·clone})$$

$$\frac{\begin{array}{c}(x{\mapsto}v)\\ \vdots\\ eval(s,a,s',v)\quad eval(s',b,s'',v')\end{array}}{eval(s,let\ x=a\ in\ b,s'',v')}(\text{e·let})$$

Figure 10. Natural Deduction Dynamic Semantics for imp_ς (alternative and additional rules).

conclusions of the rules, according to the practice of natural deduction style—see *e.g.* rules (e·call) and (e·let). It is worth noticing that we do not need to introduce the well-formedness judgements for stores and stacks (as in Figure 2), because the freshness of locally quantified variables (*eigenvariables*) is automatically provided in NDS.

A direct consequence of the NDS approach is that closures cannot be pairs $\langle method, stack\rangle$ anymore, because stacks are not "first-class" structures (such as terms or stores). The content of stacks is realised as assumptions in the proof context, which is a meta-level structure (*i.e.* of the metalan-

$$\frac{closed(b)}{wrap(b,b)}(\text{w·ground}) \qquad \frac{\begin{array}{c}(closed(y))\\ \vdots\\ y{\mapsto}v \quad y \in \mathsf{FV}(b) \quad wrap(b,\bar{b})\end{array}}{wrap(b,\bar{b}[y \mapsto v])}(\text{w·bind})$$

$$\frac{closed(a)}{closed(clone(a))}(\text{c·clone}) \qquad \frac{closed(a)}{closed(a.l)}(\text{c·call}) \qquad \frac{closed(a) \quad \begin{array}{c}(closed(x))\\ \vdots\\ closed(b)\end{array}}{closed(let\ x = a\ in\ b)}(\text{c·let})$$

$$\frac{\begin{array}{c}(closed(x_i))\\ \vdots\\ \forall i \in I \quad closed(b_i)\end{array}}{closed([l_i = \varsigma(x_i)b_i]^{i\in I})}(\text{c·obj}) \qquad \frac{closed(a) \quad \begin{array}{c}(closed(x))\\ \vdots\\ closed(b)\end{array}}{closed(a.l{\leftarrow}\varsigma(x)b)}(\text{c·upd})$$

Figure 11. Rules for $wrap$ and $closed$ judgements.

guage). Thus, we introduce the sorts of *closures* and *closure-bodies*:

$$Closure: \quad c ::= \lambda x.\bar{b} \qquad\qquad Body: \quad \bar{b} ::= b \mid \bar{b}[x{\mapsto}v]$$

where x is bound in $\bar{b}$ by $\lambda x.\bar{b}$ and $\bar{b}[x{\mapsto}v]$ (thus $\bar{b}[x{\mapsto}v]$ is like a let for results). A closure $\langle \varsigma(x)b, (x_1{\mapsto}v_1, \ldots, x_n{\mapsto}v_n)\rangle$ is then represented by:

$$\lambda x.b[x_1{\mapsto}v_1, \ldots, x_n{\mapsto}v_n] : Closure$$

where the first (outmost) abstraction λx corresponds to $\varsigma(x)$, and the following ones bind all the free variables of b to their corresponding results. For instance, the evaluation of Example 1 is now represented as $\emptyset \vdash_{\mathsf{ND}} eval(\emptyset,\ [l = \varsigma(x)x.l{\leftarrow}\varsigma(y)x].l,\ s,\ [l{=}0])$, where $s \equiv 0 \mapsto \lambda y.x[x{\mapsto}[l{=}0]]$.

Closure evaluation occurs in the method invocation (rule (e·call)). Before evaluating the inner term in a method-body, we have to add to the current proof environment all the bindings recorded in the closure (and a fresh variable representing the host object itself). This unfolding of closures is carried out by the simple auxiliary judgement $eval_b \subseteq Store{\times}Body{\times}Store{\times}Res$, defined by mutual induction with $eval$ (Figure 9).

Closure construction occurs in object creation and method updating (rules $(\text{e·obj}), (\text{e·upd}_\text{f}), (\text{e·upd}_\text{i})$). To build a closure, we have to gather from the proof context all the results associated to the free variables appearing in the method-body. This is carried out by the auxiliary judgement $wrap \subseteq Term{\times}\ Body$ (Figure 11). Informally, $\Gamma \vdash_{\mathsf{ND}} wrap(b,\bar{b})$ means that "$\bar{b}$ is a closure-body obtained by binding all the free variables of the term b to their respective results, which are in Γ". In order to keep track of free variables in terms, we need an extra judgement $closed \subseteq Term$, which formally means:

$$\Gamma \vdash_{\mathsf{ND}} closed(a) \iff \forall x \in \mathsf{FV}(a) : closed(x) \in \Gamma$$

and whose rules, completely syntax-directed, are in Figure 11. Operationally, the rules for $wrap$ allow for successively binding the free variables appearing

in a method-body $(\mathrm{w}{\cdot}\mathrm{bind})$: at each step we choose any (free) variable y in b, and bind it to the corresponding result v, as stated in Γ. If the closure $\overline{b}$ binds *all* the free variables of b, then at each step a free variable of b is marked as "closed" by a local assumption. Eventually, we have enough assumptions to be able to prove $closed(b)$, and thus we can apply the rule $(\mathrm{w}{\cdot}\mathrm{ground})$.

Notice that the closures built by $wrap$ are in general *smaller* than the original ones (Figure 3), because only the free variables are recorded in a closure (although in a non-deterministic order), not the whole current stack.

Adequacy (I). We prove now that the NDS presentation of fun_ς and imp_ς dynamic semantics corresponds faithfully to that of Sections 2.1 and 2.2. First, we establish the relationship between contexts Γ and environments S of the original setting, and between the two kinds of stores s and σ.

DEFINITION 5. *For Γ a context, S a stack, s, σ stores, we define:*

$$\begin{aligned}
\Gamma \subseteq S &\iff \forall x{\mapsto}v \in \Gamma : x{\mapsto}v \in S \quad S \subseteq \Gamma \iff \forall x{\mapsto}v \in S : x{\mapsto}v \in \Gamma \\
\gamma(S) &\iff \{x{\mapsto}S(x) \mid x \in \mathsf{Dom}(S)\} \\
s \simeq \sigma &\iff \mathsf{Dom}(s) = \mathsf{Dom}(\sigma) \wedge \\
&\qquad \forall \iota_i \in \mathsf{Dom}(s) : \gamma(S_i), closed(x_i) \vdash_{\mathsf{ND}} wrap(b_i, \overline{b_i}), \\
&\qquad \textit{where } s(\iota_i) = \lambda x_i.\overline{b_i} \textit{ and } \sigma(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle
\end{aligned}$$

For $\overline{b}$ a closure-body, let us denote by $stack(\overline{b})$ the set of bindings in $\overline{b}$, and by $body(\overline{b})$ the inner body. These functions can be defined recursively on $\overline{b}$:

$$\begin{aligned}
stack(b) &= \emptyset & stack(\overline{b}[x \mapsto v]) &= stack(\overline{b}) \cup \{x{\mapsto}v\} \\
body(b) &= b & body(\overline{b}[x \mapsto v]) &= body(\overline{b})
\end{aligned}$$

LEMMA 6.

1. *If $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow [l_i = \iota_i]^{i \in I}{\cdot}\sigma'$, then:*
   a) *$\sigma{\cdot}S \vdash_{\mathsf{AC}} \Diamond$, and $\mathsf{Dom}(\sigma) \subseteq \mathsf{Dom}(\sigma')$, and $\forall i \in I : \iota_i \in \mathsf{Dom}(\sigma')$;*
   b) *$\sigma'{\cdot}S \vdash_{\mathsf{AC}} \Diamond$;*
   c) *$\forall i \in I : \sigma'{\cdot}S_i \vdash_{\mathsf{AC}} \Diamond$, where $\sigma'(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle$.*
2. *For a closure $\langle \varsigma(x)b, S \rangle$, there exists $\overline{b}$ such that $\gamma(S), closed(x) \vdash_{\mathsf{ND}} wrap(b, \overline{b})$.*
3. *Let $\overline{b} \equiv b[x_1{\mapsto}v_1, \ldots, x_n{\mapsto}v_n]$, and let Γ be a well-formed context. Then, $\Gamma \vdash_{\mathsf{ND}} eval_b(s, \overline{b}, s', v)$ iff $\Gamma, stack(\overline{b}) \vdash_{\mathsf{ND}} eval(s, body(\overline{b}), s', v)$.*

Proof.

1. a) By structural induction on $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow [l_i = \iota_i]^{i \in I}{\cdot}\sigma'$.

   b) By structural induction on $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow [l_i = \iota_i]^{i\in I}{\cdot}\sigma'$, and point a).
   c) By point b), and inspection on the derivation of $\sigma'{\cdot}S \vdash_{\mathsf{AC}} \Diamond$.
2. By induction on $n = |S|$, using the rules $(\mathrm{w}{\cdot}\mathrm{ground})$ and $(\mathrm{w}{\cdot}\mathrm{bind})$.
3. Direction ($\Rightarrow$) can be proved by structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} eval_b(s, \bar{b}, s', v)$, while ($\Leftarrow$) by induction on n. □

Now we are ready to establish the adequacy of our NDS formulation of dynamic semantics for fun_ς and imp_ς. Let us say that Γ is a *well-formed evaluation context* if it is functional with respect to the judgement "$\mapsto$"; *i.e.* if $x{\mapsto}v, x{\mapsto}v' \in \Gamma$, then $v \equiv v'$.

PROPOSITION 7 (Adequacy of dynamic semantics).
Let Γ be a well-formed evaluation context, and $\sigma{\cdot}S \vdash_{\mathsf{AC}} \Diamond$.

1. *Let $\Gamma \subseteq S$, and $s \simeq \sigma$. If $\Gamma \vdash_{\mathsf{ND}} eval(s, a, s', v)$, then there exists σ', such that $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow v{\cdot}\sigma'$, and $s' \simeq \sigma'$;*
2. *Let $S \subseteq \Gamma$ and $\sigma \simeq s$. If $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow v{\cdot}\sigma'$, then there exists s', such that $\Gamma \vdash_{\mathsf{ND}} eval(s, a, s', v)$, and $\sigma' \simeq s'$.*

Proof.

1. By structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} eval(s, a, s', v)$. The proof is immediate for the rules $(\mathrm{e}{\cdot}\mathrm{var})$, $(\mathrm{e}{\cdot}\mathrm{obj})$, $(\mathrm{e}{\cdot}\mathrm{clone})$, $(\mathrm{e}{\cdot}\mathrm{upd_f})$ and $(\mathrm{e}{\cdot}\mathrm{upd_i})$. The $(\mathrm{e}{\cdot}\mathrm{let})$ rule requires to apply Lemma 6, points $1.a$ and $1.b$, while the $(\mathrm{e}{\cdot}\mathrm{call})$ rule points 3, $1.a$ and $1.c$ of the same lemma.
2. By structural induction on the derivation of $\sigma{\cdot}S \vdash_{\mathsf{AC}} a \rightsquigarrow v{\cdot}\sigma'$. The rules $(\mathrm{Red}{\cdot}\mathrm{Var})$ and $(\mathrm{Red}{\cdot}\mathrm{Clone})$ are addressed straightforwardly, while the remaining ones via Lemma 6: $(\mathrm{Red}{\cdot}\mathrm{Obj})$ and $(\mathrm{Red}{\cdot}\mathrm{Upd})$ need point 2, $(\mathrm{Red}{\cdot}\mathrm{Let})$ points $1.a$, $1.b$, and $(\mathrm{Red}{\cdot}\mathrm{Sel})$ points $1.a$, $1.c$, and 3. □

3.2. STATIC SEMANTICS

The term typing judgement $E \vdash_{\mathsf{AC}} a : A$ is easily rendered in NDS as $\Gamma \vdash_{\mathsf{ND}} type(a, A)$, where $type \subseteq Term \times TType$ and the proof context Γ contains typing assignments to the (free) variables, such as $x{:}A$. The judgements for well-formedness of types and subtyping are easily recovered in this setting as well, respectively as $wt \subseteq TType$ and $sub \subseteq TType \times TType$. The typing rules in natural deduction for imp_ς, are given in Figure 12; clearly, the system for fun_ς is the same without the rules $(\mathrm{t}{\cdot}\mathrm{clone})$ and $(\mathrm{t}{\cdot}\mathrm{let})$.

Notice that the well-formedness of the (distributed) typing environment is ensured by the freshness of locally quantified variables (*eigenvariables*, see *e.g.* the rules $(\mathrm{t}{\cdot}\mathrm{let})$ and $(\mathrm{t}{\cdot}\mathrm{obj})$). The premise $wt(A)$ in the rule $(\mathrm{t}{\cdot}\mathrm{var})$ ensures that non well-formed types possibly in Γ have no effect.

$$\frac{J \subseteq I \quad \forall i \in I : wt(B_i)}{sub([l_i{:}B_i]^{i\in I}, [l_i{:}B_i]^{i\in J})}(\text{sub·obj}) \qquad \frac{sub(A,B) \quad sub(B,C)}{sub(A,C)}(\text{sub·trans})$$

$$\frac{wt(A)}{sub(A,A)}(\text{sub·refl}) \qquad \frac{wt(A) \quad x{:}A}{type(x,A)}(\text{t·var}) \qquad \frac{type(a,[l_i{:}B_i]^{i\in I})}{type(clone(a),[l_i{:}B_i]^{i\in I})}(\text{t·clone})$$

$$\frac{\forall i \in I : wt(B_i) \quad \forall i \neq j : l_i \neq l_j}{wt([l_i{:}B_i]^{i\in I})}(\text{wt·obj}) \qquad \frac{type(a,[l_i{:}B_i]^{i\in I}) \quad j \in I \quad \overset{(x:[l_i:B_i]^{i\in I})}{\vdots}\ type(b,B_j)}{type(a.l_j \leftarrow \varsigma(x)b,[l_i{:}B_i]^{i\in I})}(\text{t·upd})$$

$$\frac{type(a,A) \quad \overset{(x:A)}{\vdots}\ type(b,B)}{type(let\ x = a\ in\ b,B)}(\text{t·let}) \qquad \frac{wt([l_i{:}B_i]^{i\in I}) \quad \forall i \in I : \overset{(x_i:[l_i:B_i]^{i\in I})}{\vdots}\ type(b_i,B_i)}{type([l_i = \varsigma(x_i)b_i]^{i\in I},[l_i{:}B_i]^{i\in I})}(\text{t·obj})$$

$$\frac{type(a,A) \quad sub(A,B)}{type(a,B)}(\text{t·sub}) \qquad \frac{type(a,[l_i{:}B_i]^{i\in I}) \quad j \in I}{type(a.l_j,B_j)}(\text{t·call})$$

Figure 12. Natural Deduction Static Semantics for imp_ς (fun_ς's is a subset).

Adequacy (II). Let us say that Γ is a *well-formed typing context* if it is functional *w.r.t.* the judgement ":"; *i.e.* if $x{:}A, x{:}A' \in \Gamma$, then $A \equiv A'$.

LEMMA 8. *Let Γ be a well-formed typing context, and E such that $E \vdash_{\mathsf{AC}} \Diamond$.*

1. *If $E \vdash_{\mathsf{AC}} A <: B$, then $E \vdash_{\mathsf{AC}} A$ and $E \vdash_{\mathsf{AC}} B$;*
2. *If $E \vdash_{\mathsf{AC}} a : A$, then $E \vdash_{\mathsf{AC}} A$;*
3. *If $\Gamma \vdash_{\mathsf{ND}} sub(A,B)$, then $\Gamma \vdash_{\mathsf{ND}} wt(A)$ and $\Gamma \vdash_{\mathsf{ND}} wt(B)$;*
4. *If $\Gamma \vdash_{\mathsf{ND}} type(a,A)$, then $\Gamma \vdash_{\mathsf{ND}} wt(A)$;*
5. *$\Gamma \vdash_{\mathsf{ND}} wt(A)$ if and only if $E \vdash_{\mathsf{AC}} A$;*
6. *$\Gamma \vdash_{\mathsf{ND}} sub(A,B)$ if and only if $E \vdash_{\mathsf{AC}} A <: B$.*

Proof.

1. By structural induction on the derivation of $E \vdash_{\mathsf{AC}} A <: B$.
2. By structural induction on $E \vdash_{\mathsf{AC}} a : A$, and point 1.
3. By structural induction on $\Gamma \vdash_{\mathsf{ND}} sub(A,B)$.
4. By structural induction on $\Gamma \vdash_{\mathsf{ND}} type(a,A)$, and point 3.
5. By structural induction on $\Gamma \vdash_{\mathsf{ND}} wt(A)$ and $E \vdash_{\mathsf{AC}} A$.
6. By structural induction on $\Gamma \vdash_{\mathsf{ND}} sub(A,B)$ for direction ($\Rightarrow$); by structural induction on $E \vdash_{\mathsf{AC}} A <: B$ and point 5 for direction ($\Leftarrow$). □

DEFINITION 9. *For Γ a context, E a type environment, we define:*

$$\Gamma \subseteq E \iff \forall x{:}A \in \Gamma : x{:}A \in E \qquad E \subseteq \Gamma \iff \forall x{:}A \in E : x{:}A \in \Gamma$$

PROPOSITION 10 (Adequacy of term typing).
Let Γ *be a well-formed typing context, and* $E \vdash_{\mathsf{AC}} \Diamond$.

1. *If* $\Gamma \subseteq E$, *and* $\Gamma \vdash_{\mathsf{ND}} type(a, A)$, *then* $E \vdash_{\mathsf{AC}} a : A$;
2. *If* $E \subseteq \Gamma$, *and* $E \vdash_{\mathsf{AC}} a : A$, *then* $\Gamma \vdash_{\mathsf{ND}} type(a, A)$.

Proof.

1. By structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} type(a, A)$. The proof is straightforward for rules $(\mathrm{t{\cdot}var})$, $(\mathrm{t{\cdot}clone})$ and $(\mathrm{t{\cdot}call})$. The remaining rules need some of the auxiliary properties collected in Lemma 8: $(\mathrm{t{\cdot}let})$ and $(\mathrm{t{\cdot}upd})$ require the point 2, $(\mathrm{t{\cdot}obj})$ point 4, and $(\mathrm{t{\cdot}sub})$ point 6.
2. By structural induction on the derivation of $E \vdash_{\mathsf{AC}} a : A$. The rules $(\mathrm{Val{\cdot}Clone})$ and $(\mathrm{Val{\cdot}Sel})$ are addressed immediately, while the other ones through Lemma 8: $(\mathrm{Val{\cdot}Let})$, $(\mathrm{Val{\cdot}Upd})$ and $(\mathrm{Val{\cdot}Obj})$ need the point 2, $(\mathrm{Val{\cdot}Sub})$ point 6, and $(\mathrm{Val{\cdot}Var})$ points 2 and 5. □

3.3. Result Typing and Subject Reduction

The result typing judgement $\Sigma \models v : A$ is translated as $\Gamma \vdash_{\mathsf{ND}} res(\Sigma, v, A)$, where $res \subseteq SType \times Res \times TType$, and $SType$ is the sort of store types, *i.e.* finite maps from locations to method types, as stated in Section 2.3. The intended meaning of $\Gamma \vdash_{\mathsf{ND}} res(\Sigma, v, A)$ is that the result v is given the type A, using the types assigned to locations by the store type Σ. Due to the correspondence with stores (which are not internalised), and differently from typing environments, it is not possible to distribute in the context Γ the content of store types.

The store compatibility $\Sigma \models \sigma$ is rendered as $comp \subseteq SType \times Store$: if $\Gamma \vdash_{\mathsf{ND}} comp(\Sigma, s)$, then the content of each location in the store s can be given the type indicated by Σ. The relation $ext \subseteq SType \times SType$ represents the original extension relation $\Sigma' \geq \Sigma$ over store types. The simple rules for res, $comp$, ext are collected in Figure 13.

Notice that we do not need to represent explicitly the well-formedness of store types, because this property is managed implicitly (however, we must check that store types are well-formed types, see $wt([l_i{:}\Sigma_2(\iota_i)]^{i \in I})$ in the premise of the rule $(\mathrm{t{\cdot}res})$).

On the other hand, there is no "stack typing" judgement, since stacks and type environments have vanished in the proof context. This information needs to be recovered at the metalevel: in the statement of Subject Reduction, we will require explicitly that variables and their results have coherent types.

Finally, we have to add a judgement for typing closure-bodies, *i.e.* $type_b \subseteq SType \times Body \times TType$ (see Figure 13). The intended meaning of $\Gamma \vdash_{\mathsf{ND}} type_b(\Sigma, \overline{b}, A)$ is that the closure-body $\overline{b}$ (fetched from some location of a store s, compatible with Σ) has type A. The judgement $type_b$ plays a role

$$\frac{\forall i \in I : \Sigma_1(\iota_i) \equiv [l_i{:}\Sigma_2(\iota_i)]^{i\in I} \quad wt([l_i{:}\Sigma_2(\iota_i)]^{i\in I}) \quad \iota_i \in \mathsf{Dom}(\Sigma)}{res(\Sigma, [l_i = \iota_i]^{i\in I}, [l_i{:}\Sigma_2(\iota_i)]^{i\in I})}(\text{t·res})$$

$$\frac{\mathsf{Dom}(s) = \mathsf{Dom}(\Sigma) \quad \forall \iota_i \in \mathsf{Dom}(s) : s(\iota_i) \equiv \lambda x.\overline{b} \quad \overset{(x:\Sigma_1(\iota_i))}{\overset{\vdots}{type_b(\Sigma, \overline{b}, \Sigma_2(\iota_i))}}}{comp(\Sigma, s)}(\text{t·comp})$$

$$\frac{\mathsf{Dom}(\Sigma) \subseteq \mathsf{Dom}(\Sigma') \quad \forall \iota \in \mathsf{Dom}(\Sigma) : \Sigma'(\iota) = \Sigma(\iota)}{ext(\Sigma', \Sigma)}(\text{t·ext})$$

$$\frac{type(b, A)}{type_b(\Sigma, b, A)}(\text{t·ground}) \quad \frac{res(\Sigma, v, A) \quad \overset{(y:A)}{\overset{\vdots}{type_b(\Sigma, \overline{b}, B)}}}{type_b(\Sigma, \overline{b}[y \mapsto v], B)}(\text{t·bind})$$

Figure 13. Natural Deduction Typing for results.

similar to that of $eval_b$: it unravels the local bindings recorded in closure-bodies. More precisely, we first add to the current proof environment (via the rule $(t{\cdot}bind)$) all the bindings recorded in the body (the judgement res is used, in turn, for typing the results found there); then, the inmost body can be typed using the plain $type$ judgement (via the rule $(t{\cdot}ground)$).

Adequacy (III). We address now the adequacy of the NDS result typing.

LEMMA 11. *Let Γ be a well-formed typing context, and $\overline{b}$ the closure-body $b[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. Then $\Gamma \vdash_{\mathsf{ND}} type_b(\Sigma, \overline{b}, A)$ can be derived if and only if there exist $A_1, \ldots, A_n$ such that:*

i) $\Gamma, y_1{:}A_1, \ldots, y_n{:}A_n \vdash_{\mathsf{ND}} type(b, A)$, and

ii) $\Gamma, y_1{:}A_1, \ldots, y_i{:}A_i \vdash_{\mathsf{ND}} res(\Sigma, v_{i+1}, A_{i+1})$ for every $i \in [0, n{-}1]$.

Proof. Direction ($\Rightarrow$) can be proved by structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} type_b(\Sigma, \overline{b}, A)$; direction ($\Leftarrow$) by induction on n. □

PROPOSITION 12 (Adequacy of result typing). *Let Γ be a well-formed typing context, E such that $E \vdash_{\mathsf{AC}} \Diamond$, s and σ such that $s \simeq \sigma$, and Σ a store type such that $\Sigma \vdash_{\mathsf{AC}} \Diamond$ and $\Gamma \vdash_{\mathsf{ND}} wt(\Sigma_1(\iota_i))$ for all $\iota_i \in \mathsf{Dom}(\Sigma)$.*

1. *If $\Gamma \vdash_{\mathsf{ND}} res(\Sigma, v, A)$ and $\Gamma \vdash_{\mathsf{ND}} comp(\Sigma, s)$, then $\Sigma \models v{:}A$ and $\Sigma \models \sigma$;*
2. *If $\Sigma \models v{:}A$, $\Sigma \models \sigma$ and $\mathsf{Dom}(\Sigma){=}\mathsf{Dom}(\sigma)$, then $\Gamma \vdash_{\mathsf{ND}} res(\Sigma, v, A)$ and $\Gamma \vdash_{\mathsf{ND}} comp(\Sigma, s)$.*

Proof.

1. By inspection on the hypothetical derivations. It is immediate to derive $\Sigma \models v{:}A$ using rule (Res). On the other hand, apply Lemma 11 ($\Rightarrow$) and then the soundness of term typing (point 1) for concluding $\Sigma \models \sigma$.

2. By inspection on the hypothetical derivations. We first deduce $\Gamma \vdash_{\mathsf{ND}} res(\Sigma, v, A)$ using rule $(\mathrm{t{\cdot}res})$, then apply the completeness of term typing (point 2) and Lemma 11 ($\Leftarrow$) for concluding $\Gamma \vdash_{\mathsf{ND}} comp(\Sigma, s)$. □

Subject Reduction. Now we can state and prove Subject Reduction for both fun_{ς} and imp_{ς}, using their NDS presentation. In stating the theorem, we have to require a coherence between types and results associated to the variables in the proof derivation context Γ; this is essentially equivalent to the "stack typing" judgement of (Abadi and Cardelli, 1996).

THEOREM 13 (Subject Reduction in NDS).
Let Γ be a well-formed typing and evaluation context, and Σ a store type such that, for all x, w, B: if $x{\mapsto}w \in \Gamma$ and $x{:}B \in \Gamma$, then $\Gamma \vdash_{\mathsf{ND}} res(\Sigma, w, B)$.

If $type(a, A)$, $eval(s, a, t, v)$, and $comp(\Sigma, s)$ are derivable from Γ, there exist a type A^+ and a store type Σ^+, such that $res(\Sigma^+, v, A^+)$, $ext(\Sigma^+, \Sigma)$, $comp(\Sigma^+, t)$, and $sub(A^+, A)$ are derivable from Γ.

Proof. By structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} eval(s, a, t, v)$; see Appendix A.1. □

3.4. Coinductive Result Typing and Subject Reduction

In this subsection we present an alternative and novel formulation of the typing system for results, by taking advantage of a further proof-theoretical tool provided by modern type theories, *i.e. coinduction*.

As mentioned in Section 2, the typing of results is not trivial because of potential circular structures in stores. The solution adopted in (Abadi and Cardelli, 1996) is to use store types, which assign to each location a fixed type, consistent with its content.

However, store types are list structures, which do not fit neatly in general-purpose, non-substructural proof assistants. In practice, this means that the handling of store types makes statements and proofs of metatheoretical properties (such as Subject Reduction) even more complex. It is therefore natural to ask whether, and when, is possible to get rid of these extra structures.

It turns out that we can always recover the types for a given result by *corecursively* looking at its structure and the content of all the locations it refers to, without the need of store types. To capture this process, we propose here a system for result typing, which possibly admits non well-founded, "circular" derivations. The typing system will have a *coinductive* rule, that is a rule where the conclusion is locally discharged in the assumptions. Using this rule, we can build types for results just by visiting the store and following the pointers it contains. The idea of using coinductive rules for typing goes back to (Milner and Tofte, 1991), but actually we have been inspired by modern type theories, such as $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, where coinduction is natively provided.

$$\frac{wt([l_i{:}B_i]^{i\in I}) \quad \forall i \in I : s(\iota_i) \equiv \lambda x.\overline{b}_i \qquad \begin{array}{c}(cores(s,[l_i=\iota_i]^{i\in I},[l_i{:}B_i]^{i\in I}))\\(x{:}[l_i{:}B_i]^{i\in I})\\\vdots\\ cotype_b(s,\overline{b}_i,B_i)\end{array}}{cores(s,[l_i=\iota_i]^{i\in I},[l_i{:}B_i]^{i\in I})}(\mathrm{t{\cdot}cores})$$

$$\frac{type(b,B)}{cotype_b(s,b,B)}(\mathrm{t{\cdot}coground}) \qquad \frac{cores(s,v,C) \quad \begin{array}{c}(y{:}C)\\\vdots\\ cotype_b(s,\overline{b},B)\end{array}}{cotype_b(s,\overline{b}[y\mapsto v],B)}(\mathrm{t{\cdot}cobind})$$

Figure 14. Coinductive Natural Deduction Result Typing for fun_{ς}.

The coinductive result typing system consists of two predicates $cores \subseteq Store \times Res \times TType$ and $cotype_b \subseteq Store \times Body \times TType$, with only three rules (Figure 14). The intended meaning of $\Gamma \vdash_{\mathsf{ND}} cores(s,v,A)$ is that the result v, containing pointers to the store s, can be given the type A. Similarly, $\Gamma \vdash_{\mathsf{ND}} cotype_b(s,\overline{b},B)$ means that the closure-body $\overline{b}$, where $\lambda x.\overline{b}$ is fetched in some location of the store s, has type B.

Notice that in the rule $(\mathrm{t{\cdot}cores})$ the conclusion is *discharged* as a local assumption, which plays the role of the "coinductive hypothesis" and makes the system coinductive. More precisely, the idea is that, in order to check whether a result $v \equiv [l_i{=}\iota_i]^{i\in I}$ can be given a type $[l_i{:}B_i]^{i\in I}$, we have to check that, for every $i \in I$, each method closure $\lambda x.\overline{b}_i$ (pointed to by ι_i) can be given the type $[l_i{:}B_i]^{i\in I} \Rightarrow B_i$. This means that we have to type the body of the method in a context extended with the types of the "self" variable x, and of each bound variable y in the closure. The type of the former is the same of the host object—hence the assumption $x{:}[l_i{:}B_i]^{i\in I}$. Bound variables in closures are associated to results, thus their type can be inferred by using $cores$ (co)recursively (see rule $t{\cdot}cobind$). During this process, due to pointer loops in the store, it may happen that we end up with the result v we started from. In this case, we can stop the typing deduction using the type we are trying to assign to v; to this end, the assertion we are proving has to be *assumed* in the hypotheses. Notice that the application of the coinductive hypothesis is always "guarded", because it is discharged in the subderivation of a different predicate ($cotype_b$), and thus at least one rule must be used.[3]

[3] Formally, the set-theoretical meaning of $cores$ is the greatest fixed point of an operator induced by the rules in Figure 14. In presence of coinductive hypotheses, the existence of such greatest fixed point is not trivial, because the operator may be not monotone. However, in our case the operator is monotone (and hence the definition is sound) because the application of the coinductive hypothesis is always guarded. For further details about coinductive proof systems and guarded induction, see (Crole, 1998; Giménez, 1995).

EXAMPLE 14. Let us recall the store with a loop of Example 1, obtained by the evaluation of a term with imperative update:

$$\emptyset \vdash_{\mathsf{ND}} eval(\emptyset,\ [l = \varsigma(x)x.l{\leftarrow}\varsigma(y)x].l,\ s \equiv 0 \mapsto \lambda y.x[x{\mapsto}[l{=}0]],\ [l{=}0])$$

Then, the result $[l{=}0]$, pointing into the store s, can be given the type $[l{:}[\,]]$ "by guarded induction", as follows:[4]

$$\dfrac{\dfrac{(cores(s,\ [l=0],\ [l{:}[\,]]))_{(1)} \quad \dfrac{\Delta_{x,2}}{cotype_b(s,\ x,\ [\,])}(\text{t}{\cdot}\text{coground})}{cotype_b(s,\ x[x{\mapsto}[l{=}0]],\ [\,])}(\text{t}{\cdot}\text{cobind})(2)}{cores(s,\ [l=0],\ [l{:}[\,]])}(\text{t}{\cdot}\text{cores})(1)$$

where $\Delta_{z,n}$ stands for the derivation:

$$\dfrac{\dfrac{(z{:}[l{:}[\,]])_{(n)}}{type(z,\ [l{:}[\,]])}(\text{t}{\cdot}\text{var}) \quad [l{:}[\,] <{:}\ [\,]}{type(z,\ [\,])}(\text{t}{\cdot}\text{sub})$$

On the other hand, fixed the store $t \equiv 0{\mapsto}\lambda x.x;\ 1{\mapsto}\lambda y.x[x{\mapsto}[l{=}0]]$, we can give the result $[m{=}1]$ the type $[m{:}[\,]]$ without using the circular assumption (notice the different interdependence between the predicates $cores$ and $cotype_b$, in the case):

$$\dfrac{\dfrac{\dfrac{\Delta_{x,2}}{cotype_b(t,\ x,\ [\,])}(\text{t}{\cdot}\text{coground})}{cores(t,\ [l=0],\ [l{:}[\,]])}(\text{t}{\cdot}\text{cores})(2) \quad \dfrac{\Delta_{x,1}}{cotype_b(t,\ x,\ [\,])}(\text{t}{\cdot}\text{coground})}{cotype_b(t,\ x[x{\mapsto}[l{=}0]],\ [\,])}(\text{t}{\cdot}\text{cobind})(1)}{cores(t,\ [m=1],\ [m{:}[\,]])}(\text{t}{\cdot}\text{cores})$$

□

It is important to point out that the types which can be inferred using the coinductive approach *coincide* with those given using store types. In other words, a result v can be given a type A in a store s with the coinductive typing system of Figure 14, if and only if, for some store type Σ compatible with s, v can be given the same type A using the typing system of Figure 13. Hence, the use of coinduction can be seen as a way for *internalising* store types within the structure of typing proofs.

In particular, for a fixed store, different typings given by different store types, correspond to structurally different derivations built using $cores$.

EXAMPLE 15. Let us consider the following evaluation of a nested object:

$$\emptyset \vdash_{\mathsf{ND}} eval(\emptyset,\ [m = \varsigma(x)[l = \varsigma(y)y]],\ t \equiv 0 \mapsto \lambda x.[l = \lambda y.y],\ [m{=}0])$$

[4] As usual, local hypotheses are indexed with the rules they are discharged by.

We can show that the result $[m{=}0]$ can be given two different types, $[m{:}[\,]]$ and $[m{:}[l{:}[\,]]]$, which are even not comparable through subtyping:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta_{y,1}}{type([l=\lambda y.y],\ [l{:}[\,]])}(\mathrm{t{\cdot}obj})(1) \qquad [l{:}[\,]] <: [\,]}{type([l=\lambda y.y],\ [\,])}(\mathrm{t{\cdot}sub})}{cotype_b(t,\ [l=\lambda y.y],\ [\,])}(\mathrm{t{\cdot}coground})}{cores(t,\ [m=0],\ [m{:}[\,]])}(\mathrm{t{\cdot}cores})$$

$$\dfrac{\dfrac{\dfrac{\Delta_{y,1}}{type([l=\lambda y.y],\ [l{:}[\,]])}(\mathrm{t{\cdot}obj})(1)}{cotype_b(t,\ [l=\lambda y.y],\ [l{:}[\,]])}(\mathrm{t{\cdot}coground})}{cores(t,\ [m=0],\ [m{:}[l{:}[\,]]])}(\mathrm{t{\cdot}cores}) \qquad \square$$

Summing up, since we do not need store types and all related machinery, the coinductive typing system for results is very compact and quite simpler than the original one (compare Figure 14 with Figures 8 and 13).

Adequacy (IV). The adequacy of the coinductive result typing, with respect to the original system of Section 2.3, is not trivial, since we use coinduction and do not have (explicit) store types. We address this issue by establish a relationship between our NDS presentations of result typing with store types and with coinduction.

LEMMA 16. *Let Γ be a well-formed typing context.*

1. *Let $\overline{b} \equiv b[x_1{\mapsto}\, v_1, \ldots, x_n{\mapsto} v_n]$. Then $\Gamma \vdash_{\mathsf{ND}} cotype_b(s, \overline{b}, A)$ if and only if there exist $A_1, \ldots, A_n$ such that $\Gamma, y_1{:}A_1, \ldots, y_n{:}A_n \vdash_{\mathsf{ND}} type(b, A)$, and for all $i \in [0, n{-}1]$: $\Gamma, y_1{:}A_1, \ldots, y_i{:}A_i \vdash_{\mathsf{ND}} cores(s, v_{i+1}, A_{i+1})$.*
2. *If $\Gamma \vdash_{\mathsf{ND}} res(\Sigma, v, A)$ and $\Gamma \vdash_{\mathsf{ND}} comp(\Sigma, s)$, then $\Gamma \vdash_{\mathsf{ND}} cores(s, v, A)$.*

Proof.

1. Direction ($\Rightarrow$) can be proved by structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} cotype_b(s, \overline{b}, A)$; direction ($\Leftarrow$) by induction on n.
2. By coinduction. We have to prove that, for $\iota_i \in \pi_2(\pi_i(v))$, if $\Gamma, x{:}A \vdash_{\mathsf{ND}} type_b(\Sigma, s(\iota_i), A_i)$ then $\Gamma, cores(s, v, A); x{:}A \vdash_{\mathsf{ND}} cotype_b(\Sigma, s(\iota_i), A_i)$. This can be proved by structural induction on the derivation of $\Gamma, x{:}A \vdash_{\mathsf{ND}} type_b(\Sigma, s(\iota_i), A_i)$, and using the coinductive hypothesis $cores(s, v, A)$, whose application is guarded by constructors $(\mathrm{t{\cdot}cores})$, $(\mathrm{t{\cdot}cobind})$. $\square$

In the following, for a result v, we will denote by $\sigma_{|v}$ the fragment of σ reachable by following ((co)recursively) the references in v, and by Σ_I the fragment of Σ whose domain is restricted to $\{\iota_i \mid i \in I\}$.

PROPOSITION 17 (Adequacy of coinductive result typing).
Let Γ be a well-formed typing context, and s, σ such that $s \simeq \sigma$.

1. *If $\Gamma \vdash_{\mathsf{ND}} cores(s, v, A)$, then there exists a store type Σ, such that $\Sigma \models v : A$ and $\Sigma \models \sigma_{|v}$.*
2. *If $\Sigma \models v : A$, $\Sigma \models \sigma$, $\mathsf{Dom}(\Sigma)$=$\mathsf{Dom}(\sigma)$ and $\Gamma \vdash_{\mathsf{ND}} wt(\Sigma_1(\iota_i))$ for all $\iota_i \in \mathsf{Dom}(\Sigma)$, then $\Gamma \vdash_{\mathsf{ND}} cores(s, v, A)$.*

Proof.

1. By inspection on the hypothetical derivation. Let $v \equiv [l_i{=}\iota_i]^{i \in I}$. The store type Σ is built step by step as follows. First, the fragment Σ_I is immediately determined by looking at v; then, Σ_I is extended by looking for the results contained in the closure-bodies pointed to by v, and proceeding (co)recursively. Let Σ be the store type obtained in this way. By Lemma 16.1 ($\Rightarrow$), and soundness of term typing (Lemma 10.1), we derive $\Sigma \models \sigma_{|v}$.
2. By inspection on the hypothetical derivations. First apply the completeness of inductive result typing (Lemma 10.4), then the completeness of term typing (Lemma 10.2), and finally use Lemmas 16.1 and 16.2. □

Subject Reduction with coinductive result typing. As we have shown, the coinductive typing system for results can be used without loss of generality in place of the more complex system based on store types. A natural question is now whether the coinductive system can be used for further simplifying the statement and proof of Subject Reduction (Theorem 13), getting rid of store types. The answer is yes, but only for fun_{ς}, and not for imp_{ς}.

The problem is that in a store-based semantics, Subject Reduction regards not only the types of the starting term and its resulting value, but also the types of starting and resulting stores. Store types represent exactly this information, and in Theorem 13 the store type of the resulting store is an extension of the starting one. This means that the types given to locations must not change during the computation, and when a new location is allocated, its type is decided once and forever. This "type persistence" of locations cannot be ensured without store types in presence of in-place updates, as in imp_{ς}, because in a given store, the very same result can be given different (and even not comparable) types, as in Example 15. Thus, when a closure is overwritten by a new one, the typing information we can recover coinductively from the new content may be different from that of the old content, even if the new closure can be given a type compatible with the old one. Store types circumvent this issue by fixing a single "reference" type both for the old and the new closures.

This difficulty (which actually is not proper of the coinductive typing system, but is intrinsic to the Subject Reduction property) does not arise when locations are never overwritten, because in this case the type information we

can recover from the store does not change. This happens for $\mathsf{fun}\varsigma$, where a method update allocates a *new* location without erasing the old closure, and thus the type information after an update is the same as before the update. For these reasons, it is convenient to work with the coinductive system for result typing, for proving properties regarding stores (such as Subject Reduction) for $\mathsf{fun}\varsigma$: actually, it allows us to build both recursive *and* corecursive derivations, thus providing an expressive and powerful proof tool.

Hence, for $\mathsf{fun}\varsigma$, we can state Subject Reduction without mentioning store types at all; all the typing information about a store is carried by the store itself. Of course, similarly to Theorem 13, we have to require explicitly the coherence between types and results associated to variables.

THEOREM 18 (Subject Reduction with coinductive result typing).
Let Γ *be a well-formed typing and evaluation context such that, given* x, w, B*: if* $x{\mapsto}w \in \Gamma$ *and* $x{:}B \in \Gamma$*, then* $\Gamma \vdash_{\mathsf{ND}} cores(s, w, B)$.

For a *a term of* $\mathsf{fun}\varsigma$*, if* $type(a, A)$*, and* $eval(s, a, t, v)$ *are derivable from* Γ*, there exists a type* A^+*, such that* $cores(t, v, A^+)$*, and* $sub(A^+, A)$ *are derivable from* Γ.

Proof. By structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} eval(s, a, t, v)$; see Appendix A.2. □

4. Formalization of $\mathsf{fun}\varsigma$ and $\mathsf{imp}\varsigma$ in CC$^{\text{(Co)Ind}}$

In this section, we discuss the encoding in CC$^{\text{(Co)Ind}}$ of the syntax, dynamic semantics, term and result typing for both $\mathsf{fun}\varsigma$ and $\mathsf{imp}\varsigma$. For definiteness, we work in the Coq V7.3 implementation of CC$^{\text{(Co)Ind}}$, albeit the methodology we follow can be applied in any similar Logical Framework.

Although the presentations given in Section 3 are simpler than the original systems, their formalization in CC$^{\text{(Co)Ind}}$ is still a complex task, because we have to face some subtle details which are left "implicit" on paper.

4.1. Formalization of the Syntax

Let us consider the syntax of $\mathsf{imp}\varsigma$, as that of $\mathsf{fun}\varsigma$ is just a subset. Since these calculi feature binders, we choose to represent them by means of *second-order abstract syntax*, or *weak HOAS* (Miculan, 1997; Honsell et al., 2001a):

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

Weak HOAS differs from "full" HOAS (Pfenning and Elliott, 1988), because in the latter object-level variables are considered as term placeholders at the metalevel, thus disappearing from the encoding. For instance, in full HOAS *let* should be represented as `let:Term->(Term->Term)->Term`. Due to its well-known advantages, full HOAS is the encoding methodology of choice in non-inductive logical frameworks, such as the Edinburgh LF and derivatives (*e.g.*, Twelf), but it does not fit well in inductive logical frameworks, such as CC$^{(Co)Ind}$. The problem is that since `Term` is an inductive type, a functional argument (of type `Term->Term`) can be defined by recursion or case analysis over `Term`. In this way, one could introduce *exotic terms* (Despeyroux et al., 1995), *i.e.* CC$^{(Co)Ind}$ terms not corresponding to any expression of $\mathsf{imp}\varsigma$. Exotic terms jeopardize the adequacy of encodings, and therefore they would have to be ruled out by extra "well-formedness" judgements, which in turn would complicate the whole encoding.

Using weak HOAS, we keep the advantage of using a metalevel abstraction, but we prevent the definition of functional arguments by recursion on the inductive type `Term`. Therefore we replace the domain of functional arguments with a parameter `Var`. The only terms which can inhabit `Var` are the variables of the metalanguage. One may think of `Var` as a set of constants, namely the real *names* of the object variables ($x_1, x_2, x_3, \ldots$). These constants are ranged over by variables of the metalanguage. Metalevel abstractions can be used for locally declaring new, fresh variables names, by introducing corresponding metalevel variables of type `Var`. The fact that `Var` is a parameter implies that if `x:Var` appears in a term `b`, then `b` must necessarily be of the form `b'(var(x))`, in which the object variable `x` occurs only under the constructor `var`. It is important to notice that the parametricity of `Var` makes it impossible to distinguish between variables, thus providing α-equivalence for free: `(let a [x:Var]b(x))` is automatically equal to `(let a [y:Var]b(y))`.[5]

However, weak HOAS does not cater for *substitution* of terms for variables (differently from full HOAS). Far for being a problem, this fits perfectly the needs for encoding the store-based semantics of $\mathsf{fun}\varsigma$ and $\mathsf{imp}\varsigma$, which use closures instead of substitutions of terms for variables.

Finally, notice that the constructor `var` is declared as a coercion, thus it may be omitted in the following; further, labels (*i.e.* names of methods) are encoded as natural numbers.

Objects are represented by an ad hoc list-like type `Obj`. An alternative definition could use directly the polymorphic lists of Coq library, as follows:

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

[5] A consequence of this is that α-equivalence is not a provable property of the object language in the logical framework, but it can be proved outside the LF (a meta-meta-property).

However, this definition would not allow to define some fundamental functions required to complete the formalization (such as, for example, the occurrence of variables "$\in$"): although these functions are definable by recursion on the structure of terms in our formalization, using polymorphic lists their specification would be recognized as "unguarded".

Adequacy of the encoding (I). The adequacy of the syntax encoding can be established using the arguments of the weak HOAS paradigm; see *e.g.* (Miculan, 1997; Miculan, 2001b; Honsell et al., 2001a). A complete treatment of these techniques is out of the scope of this paper; we recall briefly the basic ideas in the case of the syntax of terms, the other cases being similar.

Basically, the adequacy aims to establish a (compositional) bijection between object-language expressions of sort $Term$, and meta-language terms of type `Term` in *canonical form*. Usually, in standard first-order encodings of higher-order calculi, the "canonical form" is the well-known β-normal form: a term without unsolved β-redexes. This is not sufficient for weak HOAS encodings, where we need to define a notion of canonical form also for the types `Var` and `Var->Term` (whose inhabitants may appear in terms of type `Term`, due to the constructors `var` and *e.g.* `let`, respectively). Since `Var` is first-order and has no constructor, its terms in canonical form are only variables (of the metalanguage). This means also that terms in `Var` have no structure, and hence cannot be destructed by `Case`s. Then, we can say that the canonical terms of type `Var->Term` are always abstractions `[x:Var]`t, where t itself is canonical (in `Term`).[6]

For $X = \{x_1, \ldots, x_n\}$ a finite set of variables, let us define:

$$\begin{aligned} \mathit{Term}_X &\triangleq \{a \mid \mathsf{FV}(a) \subseteq X\} \\ \Xi_X &\triangleq x_1 : \mathtt{Var}, \ldots, x_n : \mathtt{Var} \\ \mathtt{Term}_X &\triangleq \{t \mid \Xi_X \vdash_{\mathsf{CC}} t : \mathtt{Term}, t \text{ canonical}\} \end{aligned}$$

Then, following (Miculan, 1997; Honsell et al., 2001a), it is easy to define two *encoding* and *decoding* functions:

$$\epsilon_X : \mathit{Term}_X\mathtt{->Term}_X \qquad \delta_X : \mathtt{Term}_X\mathtt{->}\mathit{Term}_X$$

such that the following property holds.

PROPOSITION 19 (Adequacy of syntax encoding). *For X a finite set of variables, ϵ_X, δ_X are* compositional *bijections, in the sense that if $a \in \mathit{Term}_{X,x}$ and $b \in \mathit{Term}_X$, then $\epsilon_X(a\{b/x\}) = \epsilon_{X,x}(a)\{\epsilon_X(b)/\,\mathtt{(var\ x)}\}$.*

Proof. By structural induction on terms of sort $Term$, and on typings of normal forms of type `Term` (Miculan, 1997; Honsell et al., 2001a). □

[6] The normalization requires a restricted form of η-expansion, that is, `M` $\rhd$ `[x:Var](M x)`. Although an algorithm for general η-normalization in CC$^{(\mathrm{Co})\mathrm{Ind}}$ is still unknown, we think that this restricted form should be decidable.

4.2. FORMALIZATION OF DYNAMIC SEMANTICS

The judgements $eval$ and $eval_b$ are represented by two inductive predicates

```
eval   : Store->Term->Store->Res->Prop
eval_b : Store->Body->Store->Res->Prop
```

whose rules are encoded using hypothetical-general judgements *à la* Martin-Löf: since the derivation contexts of the proof systems in Figures 9 and 10 obey a stack-like allocation strategy, the assignment of results to variables can be formalized through hypothetical premises, local to sub-reductions. On the other hand, since stores cannot be distributed in the proof environment, they are represented as lists of closures, where the i-th element of the list is the closure associated to the location ι_i. Each closure is simply a closure-body abstracted with respect to the "*self*" variable:

```
Definition Loc:= nat.
Definition Res: Set:= (list (Lab*Loc)).
Inductive Body: Set:= ground: Term->Body
                    | bind: Res->(Var->Body)->Body.
Definition Cls: Set:= Var->Body.
Definition Store: Set:= (list Cls).
```

(Some simple functions are needed for manipulating these structures, *e.g.* for extracting single lists from lists of pairs.) A stack is then a finite map associating each declared variable to a result; therefore, it could be represented as a functional relation,[7] or, even better, as a function `stack:Var->Res` described by a finite set of assumptions of the form "`(stack x)=v`" (where "=" is Leibniz equality). Each of such assumptions corresponds to a binding "$x{\mapsto}v$" of the context Γ; these assumptions are used in evaluating variables (rule `e_var`), and locally assumed when needed, as in the rule `e_let`:

```
Parameter stack: Var->Res.
Mutual Inductive
     eval: Store->Term->Store->Res->Prop:=
  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)
with eval_b: Store->Body->Store->Res->Prop:= ...
```

[7] Actually, in the original NDS approach within the Edinburgh LF, the `eval` judgement itself should be used for representing these bindings (Burstall and Honsell, 1990). In CC$^{(Co)Ind}$ we cannot use `eval` in place of `stack`, due to the positivity restrictions of inductive types: `eval` is inductive and the discharged hypotheses are in negative position.

In the `e_let` rule, the "hole" of `b` is filled with a fresh (*i.e.* locally quantified) variable `x` associated to `v`. This rule points out why the weak HOAS approach is well-suited for the store-based operational semantics: only substitution of variables for variables is needed, which is automatically provided by the meta-language. Similarly, the auxiliary judgement $wrap$, needed for constructing closures (Figure 11), can be formalized by an inductive predicate:

```
Parameter dummy: Var->Prop.
Inductive wrap : Term->Body->Prop:=
  w_ground: (b:Term)(closed b)->(wrap b (ground b))
| w_bind : (b:Var->Term;c:Var->Body;xl:(list Var))
           ((z:Var)(dummy z)->(wrap (b z) (c z)))->
           (y:Var;v:Res) (stack y)=(v) ->
           (isin y (b y)) ->
           (wrap (b y) (bind v c)).
```

Some explanations about `wrap` are in order. The judgement `dummy` is the usual workaround for negative occurrences of `closed`, and it is used to represent the discharged hypothesis $closed(y)$ of rule $(\mathrm{w{\cdot}bind})$. The relation $closed$ can be represented efficiently as a function `closed:Term->Prop`, defined by recursion on the structure of terms, as in (Miculan, 2001b):

```
Fixpoint closed [t:Term]: Prop:= Cases t of
 (var x)     => (dummy x) |(obj ml) => (cld_obj ml)
|(call a l) => (closed a)
|(upd a l m) => (closed a) /\
          ((x:Var)(dummy x)->(closed (m x))) | ...
with cld_obj [ml:Obj]: Prop:= Cases ml of
 (obj_nil)          => True
|(obj_cons l m nl) => (cld_obj nl) /\
          ((x:Var)(dummy x)->(closed (m x)))
```

With this definition, an assertion `(closed a):Prop` can be reduced by "`Simpl`ification" into a conjunction of similar assertions about simpler terms, which is easily dealt with using the tactics provided by Coq. The same technique can be used for defining the function `isin:Var->Term->Prop` (representing "$x \in \mathsf{FV}(a)$").

It is interesting to notice that metavariables are used with two different meanings: either as "real" variables, associated to results by `stack`, or as placeholders in the construction of closures (in this case marked as `dummy`; see discussion in Subsection 5.1). As a consequence, we cannot have both `(stack y)=(v)` and `(dummy y)` in the assumptions; indeed the assumption `(dummy z)` is about a locally quantified, fresh variable `z`.

Finally, the assumptions in the rule `w_bind` are enough to ensure also that `b:Var->Term` is a "good context" for `y`, that is, `y` does not appear free

in b. Indeed, if y appeared free in b, then it would be free also in (b z), and eventually also in the term b′, body of the method (b z), which should be proved closed after an application of w_ground. But (closed b′) would not be provable, because we would need y to be marked as dummy, which is not the case.

The remaining rules of eval are simple; both the functional and the imperative method update can be easily formalized (Ciaffaglione et al., 2003c). We discuss here only the rule for method selection, which needs eval_b for evaluating a closure body, after that the closure is retrieved from the store:

```
e_call: (s,s',t:Store;a:Term;v,w:Res;c:Cls;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_b s' (c x) t w))->
  (eval s (call a l) t w)
```

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, whose body is evaluated by eval_b after that a local variable x, denoting "self", is associated to (the implementation of) the host object. The two rules for eval_b are simple:

```
  e_ground: (s,t:Store;a:Term;v:Res)
  (eval s a t v)->(eval_b s (ground a) t v)
| e_bind: (s,t:Store;c:Var->Body;v,w:Res)
  ((y:Var)(stack y)=(v)->(eval_b s (c y) t w))->
  (eval_b s (bind v c) t w).
```

Adequacy of the encoding (II). We state now the adequacy of the formalization of dynamic semantics. As for terms, it is easy to define suitable encoding functions for the syntactic classes introduced in this subsection (locations, results, method bodies, closures and stores). We will keep denoting all these functions by ϵ_X, which map abstract entities (with free variables in X) to CC$^{\text{(Co)Ind}}$ terms of the corresponding type (with free variables in Ξ_X). As a difference, the encoding map for results does not need the $_X$ parameter. Moreover, we define the encoding map for the proof context Γ. Let $\Gamma = \{x_1{\mapsto}v_1, \ldots, x_n{\mapsto}v_n\}$ be a well-formed evaluation context; then, we define:

$$\epsilon_X(\emptyset) \triangleq \emptyset \qquad \epsilon_X(\Gamma, x{\mapsto}v) \triangleq \epsilon_X(\Gamma), \mathtt{s}\mathtt{:}\,\mathtt{(stack\ x)} = \epsilon_X(v) \qquad (1)$$

PROPOSITION 20. *Let X be a finite set of variables. Let $a \in Term_X$, $s, s' \in Store_X$, $v \in Res_X$, $\overline{b} \in Body_X$, and let Γ be a well-formed evaluation context such that, for all $x{\mapsto}v \in \Gamma$: $\{x\} \cup \mathsf{FV}(v) \subseteq X$. Then:*

$$\Gamma \vdash_{\mathsf{ND}} eval(s, a, s', v) \iff \Xi_X, \epsilon_X(\Gamma) \vdash_{\mathsf{CC}} \_ : (\mathtt{eval}\ \epsilon_X(s)\ \epsilon_X(a)\ \epsilon_X(s')\ \epsilon(v))$$
$$\Gamma \vdash_{\mathsf{ND}} eval_b(s, \overline{b}, s', v) \iff \Xi_X, \epsilon_X(\Gamma) \vdash_{\mathsf{CC}} \_ : (\mathtt{eval}\ \epsilon_X(s)\ \epsilon_X(\overline{b})\ \epsilon_X(s')\ \epsilon(v))$$

Proof. Direction ($\Rightarrow$) can be proved by mutual induction on the derivations of $eval(s, a, s', v)$ and $eval(s, \bar{b}, s', v)$; direction ($\Leftarrow$) by mutual induction on the syntax of proof terms. □

4.3. FORMALIZATION OF TERM TYPING

Term types are defined as lists of pairs of labels and types:

```
Inductive TType: Set:= mk:(list(Lab*TType))->TType.
```

In principle, this definition does not prevent to define illegal types, *i.e.* with duplicated labels. The check about their well-formedness is performed by the predicate `wt:TType->Prop`, whose definition is easy and omitted here.

The term typing judgement *type* is encoded by a judgement `type:Term->TType->Prop`. In principle, we could represent assignments of types to variables by means of assumptions of the form `(type (var x) A)`, but this would forbid to define `type` as an inductive predicate, due to the usual positivity constraints. Thus, typing of variables is represented by a specific judgement `typenv`, which acts as `type`, but restricted to variables:

```
Parameter typenv: Var->TType->Prop.
Hypothesis typenv_sub: (x:Var; A,B:TType)
          (typenv x A)->(sub A B)->(typenv x B).
```

Since `typenv` is a restricted version of `type`, it must satisfy the same properties, such as subtyping (represented by the hypothesis `typenv_sub`) and non-functionality (hence it cannot be a function of type `Var->TType`).

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

```
Mutual Inductive type: Term->TType->Prop:=
  t_sub: (a:Term;A,B:TType)
    (type a A)->(sub 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;ml:Object;l:Lab;m:Var->Term)
    (pl:(list(Lab*TType)))
    (type_obj C (obj ml) A)->
    (list_from_type A)=(pl)->
    ((x:Var)(typenv x C)->(type (m x) B))-> ...
    (type_obj C (obj (obj_cons l m ml))
                (mk (cons (l,B) pl))).
```

We omit here the encoding of `sub`, which formalizes the subtype predicate. Just notice that the formulation of the rule $(\mathrm{sub}{\cdot}\mathrm{obj})$ "on paper" (in Figure 12), hides the possibility of *permutating* the component pairs of object types, and does not address explicitly the *invariance* of types associated to identical labels. Therefore, in order to formalize this rule in a Logical Framework, it is necessary to characterize in a completely detailed way permutation and invariance. However, since the formal treatment of subtyping is neither central in the economy of the proof of Subject Reduction, nor problematic, we refer the interested reader to the discussion in (Ciaffaglione, 2003).

Adequacy of the encoding (III). As in the previous subsection, the encoding map is extended straightforwardly to types, which we will keep denoting by $\epsilon : TType\texttt{->TType}$. Moreover, we have to extend the previous definition of encoding map for proof contexts (equation (1)) to the case of type assignment:

$$\epsilon_X(\Gamma, x{:}A) \triangleq \epsilon_X(\Gamma), \mathtt{t}x\texttt{:(typenv } x\ \epsilon(A)\texttt{)}$$

Notice that, although `typenv` is a relation, the set of typing assumptions corresponding to a context Γ is always functional (*i.e.*, for each $x \in \mathrm{dom}(\Gamma)$, there is exactly one assumption $h\texttt{:(typenv } x\ \mathtt{A}\texttt{)}$).

PROPOSITION 21. *Let X be a finite set of variables. Let $a \in Term_X$, $A \in TType$, and let Γ be a well-formed typing context such that, for all $x{\mapsto}v \in \Gamma : \{x\} \cup \mathsf{FV}(v) \subseteq X$. Then:*

$$\Gamma \vdash_{\mathsf{ND}} type(a, A) \iff \Xi_X, \epsilon_X(\Gamma) \vdash_{\mathsf{CC}} \_ : (\mathtt{type}\ \epsilon_X(a)\ \epsilon(A))$$

Proof. ($\Rightarrow$) can be proved by induction on the derivation of $type(a, A)$, ($\Leftarrow$) by induction on the syntax of the (normalized) proof term. □

4.4. Formalization of Result Typing with Store Types

The judgement res of Subsection 3.3 is easily rendered by means of an inductive predicate `res`. The key issue in encoding the result typing system of Figure 13 is that we have to formalize store types, *e.g.* as lists of type pairs:

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

These lists need to be managed by means of a bunch of functions (whose definition is omitted here: see (Ciaffaglione et al., 2003c)) to play the role of store types. The system of Figure 13 is then easily rendered in CC$^{\text{(Co)Ind}}$: the encoding of ext and *comp* is straightforward; however, res needs an auxiliary inductive judgement (`resaux`), since we must carry along the whole (result) type (`A`, below) while we scan and type (the components of) results:

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

As for term typing, the adequacy of result typing can be easily established.

4.5. FORMALIZATION OF COINDUCTIVE RESULT TYPING

The judgements $cores$ and $cotype_b$ of Subsection 3.4 are rendered by means of two mutually defined *coinductive* predicates (although $cotype_b$ is intrinsically inductive). The encoding of the rules of Figure 14 needs an auxiliary coinductive judgement (`coaux`), similarly to the treatment of res above:

```
CoInductive coaux: TType->Store->Res->TType->Prop:=
t_void: (A:TType;s:Store)
  (coaux A s (nil(Lab*Loc)) (mk(nil(Lab*TType)))) |
t_step: (A,B,C:TType;s:Store;pl:(list(Lab*TType)))
  (v:Res;i:Loc;c:Cls;l:Lab) (store_nth i s)=(c)->
  ((x:Var)(typenv x C)->(cotype_b s (c x) B))->
  (coaux C s v A)->(list_from_type A)=pl->...->
  (coaux C s (cons (l,i) v) (mk (cons (l,B) pl)))
with cotype_b: Store->Body->TType->Prop:=
t_coground: (s:Store;b:Term;A:TType)
  (type b A)->(cotype_b s (ground b) A) |
t_cobind: (s:Store;b:Var->Body;A,B:TType;v:Res)
  ((x:Var)(typenv x A)->(cotype_b s (b x) B))->
  (coaux A s v A)->(cotype_b s (bind v b) B).
Definition cores: Store->Res->TType->Prop:=
  [s:Store;v:Res;A:TType] (coaux A s v A).
```

Apparently, the coinductive (discharged) hypothesis in ($t{\cdot}cores$, Figure 14) disappears from this encoding: it is not in the rule t_step. In fact, this is not the case: since `coaux` is coinductive, when we have to prove a goal of the form `(coaux ...)` we can assume it in the hypotheses using the `Cofix` tactic. So the discharged coinductive hypothesis is available "for free".

Adequacy of the encoding (IV). The adequacy of the encoding of the coinductive system for result typing is more subtle than the previous ones. Clearly, a derivation $\Gamma \vdash_{\mathsf{ND}} cores(s, v, A)$ should be represented by a coinductive term of type `(cores s v A)`, that is `(coaux A s v A)`. However, (proof) terms inhabiting coinductive types are subject to precise and stringent well-formedness conditions (Coq, 2003).

PROPOSITION 22. *Let X be a finite set of variables. Let $a \in Term_X$, $A \in Type$, $v \in Res_X$, and let Γ be a well-formed context such that, for all $x{\mapsto}w \in \Gamma : \{x\} \cup \mathsf{FV}(w) \subseteq X$. Then:*

$\Gamma \vdash_{\mathsf{ND}} cores(s, v, A) \Leftrightarrow \Xi_X, \epsilon_X(\Gamma) \vdash_{\mathsf{CC}} \_ : (\mathtt{coaux}\ \epsilon(A)\ \epsilon(s)\ \epsilon_X(a)\ \epsilon(A))$.

Proof. Direction ($\Rightarrow$) can be proved by induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} cores(s, v, A)$. In particular, when the last rule applied is $(t{\cdot}cores)$ (and the object type is not empty), the corresponding proof term is:

```
CoFix p.{p :  (coaux A s a A)
          := (t_step ? ? A s ... Q(p) ...)}
```

where `Q(p)` is the proof term of type `(x:Var)(typenv x C)->` `(cotype_b s (c x) B)`, encoding the subderivation:

$$\Gamma, x{:}A, cores(s, v, A) \vdash_{\mathsf{ND}} cotype_b(s, \overline{b}_i, B_i)$$

which exists by inductive hypothesis. The resulting circular proof term is well-formed because the occurrence of `p` is guarded by `cotype_b`. Part ($\Leftarrow$) can be proved by induction on the syntax of the (normalized) proof term. □

5. Metatheory of fun$_\varsigma$ and imp$_\varsigma$ in Coq

One of the main applications of the formalizations of fun$_\varsigma$ and imp$_\varsigma$, is the proof in Coq of fundamental properties, *e.g.* type soundness and behavioral equivalence of objects. In this section we illustrate the formal proof of the fundamental *Subject Reduction* property, already proved on paper in Section 3 in its NDS version. As mentioned in Section 2, Subject Reduction implies immediately the type soundness of type discipline.

We consider Subject Reduction for both the original and the coinductive systems for result typing of Figures 13 and 14, encoded in Coq in subsections

4.4 and 4.5, respectively. In subsection 5.1 we formalize Theorem 13, and point out some interesting aspects common to both the two versions of result typing, such as the application of the *Theory of Contexts*. In subsection 5.2 we formalize Theorem 18, and focus on the peculiar aspects and benefits provided by the coinductive result typing system.

5.1. SUBJECT REDUCTION WITH STORE TYPES

Subject Reduction in NDS (Theorem 13) can be readily formalized in Coq:

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

Notice that the proof context Γ, containing stacks and typing assertions, "disappears" from the statement: it is implicitly dealt with by the proof assistant. The proof is by structural induction on the derivation `(eval s a t v)`. Many technical lemmata about operational semantics, term and result typing have been needed. These lemmata are relatively compact and easy to prove, essentially because the object system is in natural deduction, and weak HOAS gives us α-equivalence for free (so we do not have to face the usual problems of first-order encodings, such as de Bruijn indexes or name-carrying syntax).

The drawback is that most LFs do not provide a sufficient support for reasoning about HOAS encodings (Despeyroux et al., 1995; Honsell et al., 2001b). For example, recursion and induction principles over higher-order terms (*i.e.* terms with "holes") are usually not available. An important family of properties which cannot be proved in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ are the *renaming lemmata*, such as the following preservation of typing under variable renaming:

```
Lemma rename_term: (m:Var->Term;A,B:TType;x,y:Var)
      (typenv x A) -> (typenv y A)->
      (type (m x) B) -> (type (m y) B).
```

In other words, the expressive power of LFs is limited, when it comes to reason on formalizations in (weak) HOAS. In recent years, there has been a lot of research about programming with, and reasoning about, datatypes in higher-order abstract syntax, and various approaches have been proposed; see *e.g.* (Hofmann, 1999; Fiore et al., 1999; Despeyroux and Leleu, 2001; Gabbay and Pitts, 2002; Honsell et al., 2001b; Momigliano and Ambler, 2003). Now, a general approach, in Logics, for increasing the expressive power of a logical system, is to take a suitable (and consistent) set of fundamental properties as *axioms*. This is the approach of the *Theory of Contexts* (ToC),

an axiomatization capturing some basic and natural properties of *(variable) names* and *term contexts* (Honsell et al., 2001a; Scagnetto, 2002). The Theory of Contexts consists in four axioms (indeed, axiom schemata):

freshness: (called also "unsaturation") $\forall M, \exists x : x \notin \mathsf{FV}(M)$: it captures the idea that a term cannot contain all the variables at once;

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; we need it because CC$^{\text{(Co)Ind}}$ is intuitionistic;

β-expansion: $\forall M, \forall x, \exists N(\cdot) : x \notin \mathsf{FV}(N(\cdot)) \wedge M = N(x)$;

extensionality: $\forall M(\cdot), N(\cdot), \forall x : x \notin \{\mathsf{FV}(M(\cdot), N(\cdot))\} \wedge M(x) = N(x) \Rightarrow M(\cdot) = N(\cdot)$. This means that two term contexts are equal if they are equal when applied to a fresh variable x. Together with β-expansion, extensionality allows to reason about higher-order terms.[8]

In principle, these properties can be "plugged in" an existing proof environment (such as Coq) without requiring any redesign of the system. Several case studies about untyped and simply typed λ-calculus, π-calculus, and Mobile Ambients (Miculan, 2001a; Scagnetto and Miculan, 2002; Honsell et al., 2001b; Ciaffaglione and Scagnetto, 2003) have shown that these axioms yield a smooth handling of corecursion schemata in HOAS, with a small overhead.

For these reasons, the use of ToC seems to be natural also for reasoning about fun$_\varsigma$ and imp$_\varsigma$, in Coq. In fact, the present formal development is the first application of this methodology to the object-oriented paradigm.

It turns out that the above properties are fully satisfactory for dealing with higher-order terms such as methods, closures and local declarations. For instance, the proof of the above `rename_term` requires the use of the "decidability", "β-expansion" and "extensionality" axioms.

However, in order to be useful for reasoning on fun$_\varsigma$ and imp$_\varsigma$, the "freshness" axiom has to be slightly modified with respect to its original formulation, similarly to what happens in other typed languages (Miculan, 2001a; Scagnetto and Miculan, 2002). The fact is that, in the NDS system (Section 3), (fresh) variables may have two different meanings: either associated to results (Figures 9 and 10), or just place-holders, in the construction of closures (Figure 11). In the first case the new variable is associated both to a result and a type, by the `stack` and `typenv` maps. In the second case, it is marked as `dummy`, because it does not carry any information about results. Thus, we observe a "regularity" of proof contexts: for each variable x, there is always the assumption `(typenv x A)` for some well-formed `A`, and, either `(stack x)=v` for some `v`, or `(dummy x)`. The unsaturation axiom

[8] From an operational point of view, extensionality is the restricted form of η-equivalence needed for calculating the canonical forms of weak HOAS encodings; see Section 4.1.

has to respect this regularity: a fresh variable cannot be generated without this information. This is reflected by assuming *two* forms of unsaturation:

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

Notice that we have to reflect that the new (meta)variable is actually fresh within the current context. This is obtained by the property `(fresh x xl)` (representing "x does not appear in the list of variables xl"), where the function `fresh:Var->(list Var)->Prop` is easily dealt with using the tactics provided by Coq.

A typical use of `unsat_typenv` is for proving that the type of a method-body is preserved by the closure construction:

```
Lemma wpt:(A:TType;m:Var->Term;c:Cls;x:Var;S:SType)
      (type (m x) A))->(wrap (m x) (c x)))-> ...
      (type_b S (c x) A).
```

In `unsat_res`, the premise `(res S v A)` ensures the consistency between results and types (to be associated to the same variable): it can be seen as the counterpart of the original "stack typing" judgement of Figure 8.

Finally, some remarks about the consistency of the axioms are in order. Proving the consistency of this particular version of the Theory of Contexts, within the CC$^{(Co)Ind}$ type theory, is out of the scope of this paper. The original ToC is known to be consistent with respect to (classical) higher-order logics; see (Hofmann, 1999; Bucalo et al., 2006) for a (non-trivial) construction of a model. We expect that a similar model can be defined for validating the two unsaturation axioms we have used in this paper. However, it is interesting future work checking if these models can be used for interpreting a theory of dependent types with coinduction, like CC$^{(Co)Ind}$. This task seems not quite easy, since giving a model to CC$^{(Co)Ind}$ alone is not trivial. Alternatively, one can try to give a syntactic proof of soundness (*i.e.* strong normalization) of the Calculus of (Coinductive) Constructions with the Theory of Contexts.

5.2. Subject Reduction with coinductive result typing

Subject Reduction with coinduction, stated for funς as Theorem 18, is formalized as follows, and proved by structural induction on `(eval s a t v)`:

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

It is important to notice that the use of the coinductive system of Figure 14 leads to a proof for the constructs of fun$_\varsigma$ considerably simpler than the proof (discussed in the previous subsection) based on the original system of Figure 13. In this perspective, some remarkable aspects are the following.

Concerning the ToC for fun$_\varsigma$, we need three forms of unsaturation:

```
Axiom unsat_typenv:(A:TType)(wt A) -> (xl:list Var)
  (EX x | (dummy x)/\(typenv x A)/\(fresh x xl)).
Axiom unsat_cores: (s:Store;v:Res;A:TType)(wt A) ->
  (cores s v A) -> (xl:list Var)
  (EX x | (stack x)=v/\(typenv x A)/\(fresh x xl)).
Axiom unsat_stack:(s,t:Store;a:Term;v,w:Res;c:Cls)
  (eval s a t v)\/(eval_b s (bind v c) t w) ->
  (xl:Varlist) (EX x | (stack x)=v/\(fresh x xl)).
```

The first unsaturation is the same as for imp$_\varsigma$, while the second one just a variation, because in fun$_\varsigma$ we have stores, not store types. The third form of unsaturation is new, and it allows to associate a fresh variable to a well-formed result (*i.e.*, which is obtained by a legal evaluation); this is needed to cope with the absence of store types in the proof that term evaluation preserves the type of stores (Lemma 30.(i) in Appendix A.2).

Proofs about the `cores` predicate can be carried out in Coq via the `Cofix` tactic: that is, we build infinitely regressive proofs assuming the thesis as an extra hypothesis, provided its application is guarded by introduction rules (Giménez, 1995) (see Example 14). This internal approach turns out to be very successful, because coinductive proofs do not need to be exhibited beforehand, but can be built incrementally using quite direct tactics. This corresponds to say that we do not have to exhibit a suitable store type beforehand, but we can discover the type of each location only if and when needed.

It is worthwhile noticing that in the proofs about coinductive result typing we can reuse with little effort several (patterns of) proofs (previously) developed for the original result typing. Namely, all those ones not requiring an explicit inspection on the structure of the involved store types; in such a case, simply we either keep proofs carried out by induction on the structure of results or convert them into coinductive proofs on the structure of derivations.

The benefits of the coinductive approach can be better appreciated by considering the proofs which must deal with store types. Tipically, these proofs are carried out reasoning by simultaneous induction over both the structure (and the content) of stores and store types. It is apparent that such proofs become much simpler when we have to deal just with stores.

6. Conclusion

In this paper, we have presented a case study about formal reasoning on object-based calculi with binders in Type Theory-based Logical Frameworks.

Our experiment has been carried out on both a functional and an imperative object calculus; we have worked in the Calculus of (Co)Inductive Constructions, implemented in the Coq proof assistant. As an example of application of the formalization, we have internally proved the property of Subject Reduction, for both calculi.

Our aim was to illustrate the benefits of taking as much as possible advantage of the proof theoretical techniques provided by modern type theories, such as Natural Deduction Semantics and Coinduction, in combination with Higher-Order Abstract Syntax and the Theory of Contexts.

The reformulation in Natural Deduction style of the systems defining the semantics of the calculi, has allowed to represent stacks and typing contexts by means of hypothetical premises. Therefore, in the subsequent formalization, stacks and typing contexts are implicitly dealt with by the metalanguage, and hence judgements and proofs have become fairly simpler than traditional ones in Natural Semantics. Furthermore, the use of Coinduction has suggested a novel, simpler typing system for results which does not require extra structures as store types, thus simplifying further the encoding.

Weak HOAS allows to deal with binders without having to encode neither α-equivalence (which is inherited from the metalanguage), nor substitution (which is not required by the calculi). A consequence of these choices is that closures are treated more efficiently than in the original system (although in a bit more complicated way).

In order to gain the extra expressive power required for proving Subject Reduction, we have added the axioms of the Theory of Contexts to our encoding. In our opinion, this is an acceptable price to pay, because the use of weak HOAS has a direct impact on the complexity of proofs, and in particular it allows for a simpler and smoother formal treatment of complex (meta)properties *w.r.t.* first-order techniques, as de Bruijn indexes or explicit names. On the other hand, the Theory of Contexts can be plugged in existing LFs without requiring any redesign of these metalogical systems.

From this experience, we can affirm that the methodology we have chosen is well-suited *w.r.t.* the proof practice, also in the challenging case of an object-oriented calculus, because it reduces considerably the length and the complexity of proofs. In particular, since weak HOAS does not provide automatically general substitution, this methodology seems best suited for dealing with store-oriented semantics, such as a semantics with method closures, where just a simple treatment of the α-equivalence is required.

6.1. Related work

Formalization of Object Calculi. To our knowledge, this is the first systematic formalization of the theory and meta-theory of Abadi and Cardelli's object-based calculi in Logical Frameworks based on Type Theory. The clos-

est work are (Laurent, 1997; Gillard, 2000; Hofmann and Tang, 2000), but we are not aware of formal approaches to static and dynamic semantics of object calculi with imperative features as ours.

In (Laurent, 1997), the functional calculus $Ob_{1<:\mu}$ is specified in the Centaur system using traditional first-order techniques and basic Natural Semantics; this encoding is then automatically translated in Coq, and finally the type soundness of $Ob_{1<:\mu}$ is proved in the proof assistant.

(Gillard, 2000) considers a functional ς-calculus extended with concurrent primitives, and uses de Bruijn indexes for dealing with bound variables via Gordon and Melham's approach to α-conversion for defining a generic second-order binder (like in (Norrish, 2003)). On one hand, this methodology allows for using automatic tools (such as Centaur), but on the other hand it suffers the usual drawbacks of first-order encodings. This is the reason why in the present paper we have striven for more advanced encoding techniques, aiming at a more sophisticated treatment of environment and binders. We believe that our approach pays off when it comes to prove theorems interactively, even if full automatized support is still under development.

(Hofmann and Tang, 2000) presents a formalization of Abadi and Leino's AL logic (an axiomatic semantics analogous to Hoare logic) for an imperative object-based calculus similar to $\mathsf{imp}\varsigma$, in the LEGO and PVS proof assistants. The syntax is represented using HOAS, whereas the operational semantics is not formalized directly. Instead, the assertions of AL are encoded, using a *shallow*, direct embedding in *higher-order* logic *à la* System F. Each inference rule of AL is then taken as axiom. This approach is quite different from ours. The encoding is simplified because operational semantics is not formalized, thus avoiding the need of formalizing locations and stores, but losing at the same time the possibility of proving properties such as Type Soundness and Subject Reduction. Moreover, since all the rules are taken as axioms, an external proof of soundness is needed, such as that in (Hofmann and Tang, 2001), relying on a semantic argument in presheaf categories. However, the comparison between these two approaches is interesting future work; for instance, it would be interesting to encode the AL logic using Hofmann and Tang's approach in our formalization, and to formally validate AL rules with respect to the operational semantics.

Linear Logical Frameworks. Since the explicit management of bulky list-like structures in judgements is unwieldy, one key point of the NDS approach is to delegate as much as possible the management of stacks and typing structures to the meta-level proof context. Unluckily, the structural features of Natural Deduction prevent us to internalise also the *store*. As shown by (Miller, 1994; Chirimar, 1995), stores can be neatly internalised in *linear* logical frameworks, such as Forum or LLF (Cervesato and Pfenning, 2002). However, these systems do not provide a native support for coinduction, nor they

are known, widespread and supported as intuitionistic ones. For these reasons, in this paper we have preferred to work in a more traditional intuitionistic type theory, namely $CC^{(Co)Ind}$; we leave for future work the investigation of the meta-theory of HOAS encodings in linear logical frameworks.

Another possibility is to use Felty's elegant *two-level* approach for encoding sub-structural logics within $CC^{(Co)Ind}$ (Felty, 2002), in the tradition of (McDowell and Miller, 1997). In this approach the metalanguage is used for representing the *sequents* of the logics, and all the peculiar structural rules one possibly needs. Therefore, besides the known judgements (typing, evaluation, etc), one has to introduce a further meta-judgement which represents the sequent itself. A first problem is that all the rules for the logical constructors already present at the meta-level must be replicated at the specification level; thus, the automatizing tactics of Coq would not work anymore. Another drawback is that one would get again "sequents", that is judgements crammed with lists of propositions, which are not easy to deal with. So this approach, although feasible in theory, is in contrast with the choices made in this work, aiming to exploit every feature the metalanguage gives us. We leave the formalization of ς-calculi in Felty's approach for future work.

Coinductive Typing Systems. Coinductive typing systems date back to (Milner and Tofte, 1991) in functional languages with fixpoints, whose values (closures) may be not well-founded. There are some similarities with our work, but here values are always finite entities; instead, potential loops may arise due to pointers to the store. Another distinguishing fact is that we deal with a different paradigm (*i.e.* object-oriented), which we consider at a low, implementation-oriented level. Thus, the calculus is considerably more complex than Milner-Tofte's, and extra structures (*i.e.* the store) are used to manage efficiently closures, like a compiler for a register machine would do.

The importance of having a native support for coinduction is confirmed also by Frost's implementation of Milner-Tofte's work in Isabelle (Frost, 1995). Frost reports that in the implementation in Isabelle/HOL using an impredicative, higher-order encoding of greatest fixed points, "4/5 of the work was about the management of fixed points". Moreover, bisimulations had to be provided explicitly beforehand, whereas we can build them implicitly, in due course, using specialized tactics. On the other hand, the implementation of the same object system in Isabelle/ZF using the coinductive package "reduced the work required dramatically". This different approach to coinduction has a great benefit on the interactive practice, and for this reason we aimed to take most advantage of $CC^{(Co)Ind}$ support to coinduction.

6.2. Other future work

An obvious possible future work is to experiment further with the formalization carried out so far, *e.g.* for proving other (meta)properties of fun_ς and

$\mathsf{imp}\varsigma$: behavioural equivalences of objects, or the formal equivalence between the two encodings for result typing, as stated on paper in Section 3.4.

We think that the presented approach can be applied also to other object-based calculi, *e.g.* those featuring *object extension* (Fisher et al., 1994).

A promising application of the formalizations is the *certification* of tools, such as interpreters, compilers and type-checkers. Some results in this direction, using Coq and Isabelle, are the certification of compilers for an imperative language (Bertot, 1998) and Java (Strecker, 2002). However, none of these works adopts higher-order abstract syntax for dealing with binders: we believe that the use of Natural Deduction Semantics and HOAS can simplify these advanced tasks in the case of languages with binders.

On a theoretical side, it is interesting future work to investigate how the current Theory of Contexts can be generalized to subsume uniformly the several variants used in the case studies about typed languages, such as ours or (Miculan, 2001a), where we had to modify slightly the unsaturation axiom.

Since the Theory of Contexts has been proved to be so useful, it is high time to consider seriously all the proof-theoretical issues concerning type theories. In particular, a syntactic proof of soundness (*i.e.* strong normalization) of the Calculus of (Coinductive) Constructions with the Theory of Contexts should be pursued; then, the Theory of Contexts could be internalised in the proof assistant, yielding an integrated Coq-ToC system.

Acknowledgements. The authors wish to thank Furio Honsell, Bernard Serpette, Yves Bertot and Joëlle Despeyroux, for fruitful discussions on earlier formalizations of $\mathsf{imp}\varsigma$. The authors wish to thank also the anonymous referees for their useful and precise comments, remarks and suggestions, in particular for drawing the attention to the functional object-calculus $\mathsf{fun}\varsigma$.

References

Abadi, M. and L. Cardelli: 1996, *A theory of objects*. Springer-Verlag.

Barendregt, H. and T. Nipkow (eds.): 1994, 'Proc. of TYPES', Vol. 806 of *Lecture Notes in Computer Science*.

Bertot, Y.: 1998, 'A certified compiler for an imperative language'. Technical Report RR-3488, INRIA.

Bucalo, A., M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto: 2006, 'Consistency of the Theory of Contexts'. *Journal of Functional Programming* **16**(3), 327–395.

Burstall, R. and F. Honsell: 1990, 'Operational semantics in a natural deduction setting'. In: G. Huet and G. Plotkin (eds.): *Logical Frameworks*. pp. 185–214, Cambridge University Press.

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

Cervesato, I. and F. Pfenning: 2002, 'A Linear Logical Framework'. *Information & Computation* **179**(1), 19–75.

Chirimar, J. L.: 1995, 'Proof Theoretic Approach to Specification Languages'. Ph.D. thesis, University of Pennsylvania.

Ciaffaglione, A.: 2003, 'Certified reasoning on Real Numbers and Objects in Co-inductive Type Theory'. Ph.D. thesis, Dipartimento di Matematica e Informatica, Università di Udine, Italy and INPL-ENSMN, Nancy, France.

Ciaffaglione, A., L. Liquori, and M. Miculan: 2003a, 'Imperative object-calculi in (Co)inductive type theories'. In: *Proc. of LPAR*, Vol. 2850 of *Lecture Notes in Computer Science*. Springer-Verlag.

Ciaffaglione, A., L. Liquori, and M. Miculan: 2003b, 'Reasoning on an Imperative object-calculus in Higher-Order Abstract Syntax'. In (Honsell et al., 2003), ACM.

Ciaffaglione, A., L. Liquori, and M. Miculan: 2003c, 'The Web Appendix of this paper'. University of Udine, Italy. `http://www.dimi.uniud.it/ciaffagl`.

Ciaffaglione, A. and I. Scagnetto: 2003, 'Plug and play the theory of contexts in higher-order abstract syntax'. In: *Proceedings of CoMeta*.

Coq: 2003, 'The Coq Proof Assistant 7.3'. INRIA. `http://coq.inria.fr`.

Crole, R. L.: 1998, 'Lectures on [Co]Induction and [Co]Algebras'. Technical Report 1998/12, Department of Mathematics and Computer Science, University of Leicester.

Despeyroux, J.: 1986, 'Proof of Translation in Natural Semantics'. In: *Proc. of LICS*. pp. 193–205, The ACM Press.

Despeyroux, J., A. Felty, and A. Hirschowitz: 1995, 'Higher-order syntax in Coq'. In: *Proc. of TLCA*, Vol. 905 of *Lecture Notes in Computer Science*. Springer-Verlag.

Despeyroux, J. and P. Leleu: 2001, 'Recursion over objects of functional type.'. *Mathematical Structures in Computer Science* **11**(4), 555–572.

Felty, A. P.: 2002, 'Two-Level Meta-reasoning in Coq.'. In: V. Carreño, C. Muñoz, and S. Tashar (eds.): *Proc. TPHOLs*, Vol. 2410 of *Lecture Notes in Computer Science*. pp. 198–213, Springer.

Fiore, M. P., G. D. Plotkin, and D. Turi: 1999, 'Abstract Syntax and Variable Binding'. In (Longo, 1999), pp. 193–202, IEEE Computer Society Press.

Fisher, K., F. Honsell, and J. Mitchell: 1994, 'A lambda calculus of objects and method specialization'. *Nordic Journal of Computing*.

Frost, J.: 1995, 'A Case Study of Co-induction in Isabelle'. Technical Report 359, University of Cambridge, Computer Laboratory. Revised version of CUCL 308, August 1993.

Gabbay, M. J. and A. M. Pitts: 2002, 'A New Approach to Abstract Syntax with Variable Binding'. *Formal Aspects of Computing* **13**, 341–363.

Gillard, G.: 2000, 'A formalization of a concurrent object calculus up to alpha-conversion'. In: *Proc. of CADE*. Springer-Verlag.

Giménez, E.: 1995, 'Codifying guarded recursion definitions with recursive schemes'. In: J. Smith (ed.): *Proc. of TYPES*, Vol. 996 of *Lecture Notes in Computer Science*. pp. 39–59, Springer-Verlag.

Harper, R., F. Honsell, and G. Plotkin: 1993, 'A framework for defining logics'. *Journal of ACM* **40**(1).

Hofmann, M.: 1999, 'Semantical analysis of higher-order abstract syntax'. In (Longo, 1999), pp. 204–213, IEEE Computer Society Press.

Hofmann, M. and F. Tang: 2000, 'Implementing a Program Logic of Objects in a Higher-Order Logic Theorem Prover'. In: *Proc. of TPHOLs*. pp. 268–282.

Hofmann, M. and F. Tang: 2001, 'A Higher-Order Embedding of a Logic of Objects'. Technical Report EDI-INF-RR-0033, LFCS, Univ. of Edinburgh.

Honsell, F., M. Miculan, and A. Momigliano (eds.): 2003, 'Eighth ACM SIGPLAN Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2003'. ACM.

Honsell, F., M. Miculan, and I. Scagnetto: 2001a, 'An axiomatic approach to metareasoning on systems in higher-order abstract syntax'. In: *Proc. ICALP*, Vol. 2076 of *Lecture Notes in Computer Science*. pp. 963–978.

Honsell, F., M. Miculan, and I. Scagnetto: 2001b, 'π-calculus in (Co)Inductive Type Theory'. *Theoretical Computer Science* **253**(2), 239–285.

Huisman, M.: 2001, 'Reasoning about Java programs in higer order logic with PVS and Isabelle'. Ph.D. thesis, Katholieke Universiteit Nijmegen.

Kahn, G.: 1987, 'Natural Semantics'. In: *Proc. of STACS*, Vol. 247 of *Lecture Notes in Computer Science*. pp. 22–39, Springer-Verlag.

Klein, G. and T. Nipkow: 2003, 'Verified Bytecode Verifiers'. *Theoretical Computer Science* **298**(3).

Laurent, O.: 1997, 'Sémantique Naturelle et Coq : vers la spécification et les preuves sur les langages à objets'. Technical Report RR-3307, INRIA.

Longo, G. (ed.): 1999, 'Proc. of LICS'. IEEE Computer Society Press.

Marché, C., C. Paulin-Mohring, and X. Urbain: 2004, 'The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML'. *J. Log. Algebr. Program.* **58**(1-2), 89–106.

McDowell, R. and D. Miller: 1997, 'A Logic for Reasoning with Higher-Order Abstract Syntax'. In: *Proc. 12^{th} LICS*.

Miculan, M.: 1994, 'The Expressive Power of Structural Operational Semantics with Explicit Assumptions'. In (Barendregt and Nipkow, 1994), pp. 292–320.

Miculan, M.: 1997, 'Encoding Logical Theories of Programs'. Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Italy.

Miculan, M.: 2001a, 'Developing (Meta)Theory of Lambda-calculus in the Theory of Contexts'. In: S. Ambler, R. Crole, and A. Momigliano (eds.): *Proc. of MERLIN*, Vol. 58.1 of *ENTCS*. pp. 1–22, Elsevier.

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

Miller, D.: 1994, 'A Multiple-Conclusion Meta-Logic'. In: S. Abramsky (ed.): *Proc. of LICS*. Paris, pp. 272–281.

Milner, R. and M. Tofte: 1991, 'Co-induction in relational semantics'. *Theoretical Computer Science* **87**, 209–220.

Momigliano, A. and S. Ambler: 2003, 'Multi-Level Meta-Reasoning with Higher Order Abstract Syntax'. In: *Proc. of FOSSACS*. Springer-Verlag.

Norrish, M.: 2003, 'Mechanising Hankin and Barendregt using the Gordon-Melham axioms.'. In (Honsell et al., 2003), ACM.

Pfenning, F. and C. Elliott: 1988, 'Higher-order abstract syntax'. In: *Proc. of ACM SIGPLAN '88 Symposium on Language Design and Implementation*. pp. 199–208.

Scagnetto, I.: 2002, 'Reasoning about Names In Higher-Order Abstract Syntax'. Ph.D. thesis, Dipartimento di Matematica e Informatica, Università di Udine, Italy.

Scagnetto, I. and M. Miculan: 2002, 'Ambient Calculus and its Logic in the Calculus of Inductive Constructions'. In: F. Pfenning (ed.): *Proc. of LFM*, Vol. 70.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

Self: 2003, 'The Self Programming Language'. Sun Microsystems. `http://research.sun.com/self/language.html`.

Strecker, M.: 2002, 'Formal Verification of a Java Compiler in Isabelle'. In: *Proc. of CADE*, Vol. 2392 of *Lecture Notes in Computer Science*. pp. 63–77, Springer-Verlag.

Tews, H.: 2000, 'A case study in coalgebraic specification: memory management in the FIASCO microkernel'. Technical report, TU Dresden.

Van den Berg, J., B. Jacobs, and E. Poll: 2001, 'Formal Specification and Verification of JavaCard's Application Identifier Class'. In: *Proc. of the JavaCard 2000 Workshop*.

Appendix

A. Subject Reduction Theorem

We document here the proof of Subject Reduction in Natural Deduction Semantics (Theorem 13), and with coinductive result typing (Theorem 18). We will write often $\mathcal{A}$ for $\Gamma \vdash_{\mathsf{ND}} \mathcal{A}$, and $\mathcal{B} \vdash_{\mathsf{ND}} \mathcal{A}$ for $\Gamma, \mathcal{B} \vdash_{\mathsf{ND}} \mathcal{A}$, where Γ is a well-formed (evaluation and typing) context.

LEMMA 23 (Typing system for terms).

$$
\begin{array}{ll}
(var) & : \ type(x, A) \ \Rightarrow \ \exists B : TType : \\
 & \quad (x{:}B \in \Gamma) \wedge sub(B, A) \\
(obj) & : \ type([l_i = \varsigma(x_i)b_i]^{i\in I}, A) \ \Rightarrow \ \exists B : TType : \\
 & \quad type([l_i = \varsigma(x_i)b_i]^{i\in I}, B) \wedge sub(B, A) \\
(call) & : \ type(a.l, A) \ \Rightarrow \ \exists B : TType : \\
 & \quad type(a, B) \wedge B \equiv [l{:}B_l, \ldots] \wedge sub(B_l, A) \\
(upd) & : \ type(a.l{\leftarrow}\varsigma(x)b, A) \ \Rightarrow \ \exists B : TType : \\
 & \quad type(a, B) \wedge sub(B, A) \wedge B \equiv [l{:}B_l, \ldots] \wedge \\
 & \quad x{:}B \vdash_{\mathsf{ND}} type(b, B_l) \\
(clone) & : \ type(clone(a), A) \ \Rightarrow \ \exists B : TType : \\
 & \quad type(a, B) \wedge sub(B, A) \\
(let) & : \ type(let(a, \lambda x.b), A) \ \Rightarrow \ \exists B, C : TType : \\
 & \quad type(a, C) \wedge sub(B, A) \wedge x{:}C \vdash_{\mathsf{ND}} type(b, B) \\
(bd{\cdot}weak) & : \ x{:}A \vdash_{\mathsf{ND}} type(b, C) \wedge sub(B, A) \ \Rightarrow \ x{:}B \vdash_{\mathsf{ND}} type(b, C)
\end{array}
$$

Proof. By structural induction on the (first) hypothesis. □

A.1. Subject Reduction in NDS (IMPς)

LEMMA 24 (Result typing).

$$
\begin{array}{l}
(i) \ ext(\Sigma, \Sigma) \\
(ii) \ ext(\Sigma'', \Sigma') \wedge ext(\Sigma', \Sigma) \ \Rightarrow \ ext(\Sigma'', \Sigma) \\
(iii) \ res(\Sigma, v, A) \wedge ext(\Sigma', \Sigma) \ \Rightarrow \ res(\Sigma', v, A)
\end{array}
$$

LEMMA 25 (Objects).

$$
\begin{array}{l}
(i) \ A \equiv [l_i{:}B_i]^{i\in I} \ \Rightarrow \ res((\Sigma, \iota_i{\mapsto}(A \Rightarrow B_i))^{i\in I}, [l_i = \iota_i]^{i\in I}, A) \\
(ii) \ A \equiv [l_i{:}B_i]^{i\in I} \wedge type([l_i = \varsigma(x_i)b_i]^{i\in I}, A) \wedge \\
\quad closed(x_i) \vdash_{\mathsf{ND}} wrap(b_i, \bar{b}_i)^{i\in I} \wedge comp(\Sigma, s) \ \Rightarrow \\
\quad comp((\Sigma, \iota_i{\mapsto}(A \Rightarrow B_i))^{i\in I}, (s, \iota_i{\mapsto}\lambda x_i.\bar{b}_i)^{i\in I})
\end{array}
$$

Proof. (i) By the rule (t·res). (ii) By induction on the object type A. □

LEMMA 26 (Method invocation).

$$comp(\Sigma, s) \wedge s(\iota_i) = \lambda x.\overline{b} \;\Rightarrow\; x{:}\Sigma_1(\iota_i) \vdash_{\mathsf{ND}} type_b(\Sigma, \overline{b}, \Sigma_2(\iota_i))$$

Proof. By inspection on the rule $(\mathrm{t{\cdot}comp})$. □

LEMMA 27 (Imperative method update).

$$\begin{array}{rcl}(i) & : & x{:}A \vdash_{\mathsf{ND}} type(b, B) \wedge closed(x) \vdash_{\mathsf{ND}} wrap(b, \overline{b}) \wedge \\ & & (\forall x, w, C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} res(\Sigma, w, C)) \Rightarrow \\ & & x{:}A \vdash_{\mathsf{ND}} type_b(\Sigma, \overline{b}, B) \\ (ii) & : & comp(\Sigma, s) \wedge x{:}\Sigma_1(\iota_i) \vdash_{\mathsf{ND}} type_b(\Sigma, \overline{b}, \Sigma_2(\iota_i)) \;\Rightarrow \\ & & comp(\Sigma, (s.\iota_i{\leftarrow}\lambda x.\overline{b}))\end{array}$$

Proof. (i) By structural induction on $closed(x) \vdash_{\mathsf{ND}} wrap(b, \overline{b})$. (ii) By the rule $(\mathrm{t{\cdot}comp})$ and point (i). □

LEMMA 28 (Cloning).

$$\begin{array}{rcl}(i) & : & res(\Sigma, [l_i = \iota_i]^{i\in I}, A) \;\Rightarrow \\ & & res((\Sigma, \iota'_i{\mapsto}\Sigma(\iota_i))^{i\in I}, [l_i = \iota'_i]^{i\in I}, A) \\ (ii) & : & comp(\Sigma, s) \;\Rightarrow \\ & & comp((\Sigma, \iota'_i{\mapsto}\Sigma(\iota_i))^{i\in I}, (s, \iota'_i{\mapsto}s(\iota_i))^{i\in I})\end{array}$$

Proof. (i) By induction on the result $[l_i = \iota_i]^{i\in I}$. (ii) By induction on the store type fragment $\iota'_i{\mapsto}\Sigma(\iota_i)^{i\in I}$. □

THEOREM 29 (Subject Reduction in NDS, $\mathsf{imp}\varsigma$). *Let* Γ *be well-formed:*

$$\begin{array}{l}\Gamma \vdash_{\mathsf{ND}} type(a, A) \;\wedge\; \Gamma \vdash_{\mathsf{ND}} eval(s, a, t, v) \;\wedge\; \Gamma \vdash_{\mathsf{ND}} comp(\Sigma, s) \wedge \\ (\forall x, w, C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} res(\Sigma, w, C)) \;\Rightarrow \\ \exists A^+{:}TType, \Sigma^+{:}SType : \Gamma \vdash_{\mathsf{ND}} res(\Sigma^+, v, A^+) \;\wedge \Gamma \vdash_{\mathsf{ND}} ext(\Sigma^+, \Sigma) \wedge \\ \Gamma \vdash_{\mathsf{ND}} comp(\Sigma^+, t) \;\wedge\; \Gamma \vdash_{\mathsf{ND}} sub(A^+, A)\end{array}$$

Proof. By structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} eval(s, a, t, v)$. The rules $(\mathrm{e{\cdot}call})$ and $(\mathrm{e{\cdot}bind})$ require a *mutual* structural induction argument, namely a stronger induction schema valid also for the predicate $eval_b$, which is the counterpart of $eval$ for closure-bodies.

$(\mathrm{e{\cdot}var})$. By hypothesis $type(x, A)$ and:

$$\frac{x{\mapsto}v}{eval(s, x, s, v)}(\mathrm{e{\cdot}var})$$

From Lemma 23.(var), there exists B such that $x{:}B \in \Gamma$ and $sub(B, A)$. Choose $A^+ := B$ and $\Sigma^+ := \Sigma$.

Since $x{\mapsto}v \in \Gamma$, by the fourth hypothesis of the theorem we can derive $res(\Sigma^+, v, A^+)$. We have $ext(\Sigma^+, \Sigma^+)$ by Lemma 24.(i) and $comp(\Sigma^+, s)$ by hypothesis, thus concluding.

(e·obj). By hypothesis $type([l_i = \varsigma(x_i)b_i]^{i\in I}, A)$ and:

$$\frac{\begin{array}{cc} & (closed(x_i)) \\ & \vdots \\ \forall i \in I : \iota_i \notin \mathsf{Dom}(s) & wrap(b_i, \overline{b}_i) \end{array}}{eval(s, [l_i = \varsigma(x_i)b_i]^{i\in I}, (s, \iota_i{\mapsto}\lambda x_i.\overline{b}_i)^{i\in I}, [l_i = \iota_i]^{i\in I})}(\text{e·obj})$$

By Lemma 23.(obj), there exists $[l_i{:}B_i]^{i\in I}$ such that:

$$type([l_i = \varsigma(x_i)b_i]^{i\in I}, [l_i{:}B_i]^{i\in I}) \tag{2}$$

and $sub([l_i{:}B_i]^{i\in I}, A)$.
Choose $A^+ := [l_i{:}B_i]^{i\in I}$ and $\Sigma^+ := \Sigma, (\iota_i{\mapsto}(A^+ \Rightarrow B_i))^{i\in I}$.

We have $res(\Sigma^+, [l_i = \iota_i]^{i\in I}, A^+)$ by Lemma 25.(i), and it is immediate that $ext(\Sigma^+, \Sigma)$. Then, since $comp(\Sigma, s)$ and (2), we apply the Lemma 25.(ii), thus concluding $comp(\Sigma^+, (s, \iota_i{\mapsto}\lambda x_i.\overline{b}_i)^{i\in I})$.

(e·call). By hypothesis $type(a.l_j, A)$ and:

$$\frac{eval(s, a, s', [l_i = \iota_i]^{i\in I}) \quad s'(\iota_j) = \lambda x.\overline{b}_j \quad \begin{array}{c} (x{\mapsto}[l_i = \iota_i]^{i\in I}) \\ \vdots \\ eval_b(s', \overline{b}_j, s'', v) \end{array} \quad j \in I}{eval(s, a.l_j, s'', v)}(\text{e·call})$$

By Lemma 23.$(call)$, there exists $[l_j{:}B_j, \ldots]$ such that $type(a, [l_j{:}B_j, \ldots])$ and $sub(B_j, A)$. Since $eval(s, a, s', [l_i = \iota_i]^{i\in I})$, by inductive hypothesis there exist C, Σ' such that:
(a). $res(\Sigma', [l_i = \iota_i]^{i\in I}, C)$;
(b). $ext(\Sigma', \Sigma)$;
(c). $comp(\Sigma', s')$;
(d). $sub(C, [l_j{:}B_j, \ldots])$.
Among the premises of the rule (e·call), we have $j \in I$, $s'(\iota_j) = \lambda x.\overline{b}_j$ and:

$$x{\mapsto}[l_i = \iota_i]^{i\in I} \vdash_{\mathsf{ND}} eval_b(s', \overline{b}_j, s'', v) \tag{3}$$

Moreover, we have $C \equiv [l_j{:}B_j, \ldots]$ from (d), thus $\Sigma'(\iota_j) = (C \Rightarrow B_j)$, and so, by (c) and Lemma 26:

$$x{:}C \vdash_{\mathsf{ND}} type_b(\Sigma', \overline{b}_j, B_j) \tag{4}$$

We deduce $(\forall x, w, C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} res(\Sigma', w, C))$ from the fourth hypothesis of the theorem and Lemma 24.(iii). Then, since (3), (4) and (c), we can apply the mutual induction hypothesis, thus concluding there exist A^+, Σ^+ such that $res(\Sigma^+, v, A^+)$ and $ext(\Sigma^+, \Sigma')$ and $comp(\Sigma^+, s'')$ and $sub(A^+, B_j)$. We finish by transitivity of ext (Lemma 24.(ii)) and transitivity of subtyping.

($\mathrm{e{\cdot}upd_i}$). By hypothesis $type(a.l{\leftarrow}\varsigma(x)b, A)$ and:

$$\frac{eval(s,a,s',[l_i=\iota_i]^{i\in I}) \quad \begin{array}{c}(closed(x))\\ \vdots \\ wrap(b,\bar{b})\end{array} \quad (j\in I)}{eval(s,a.l{\leftarrow}\varsigma(x)b,(s'.\iota_j{\leftarrow}\lambda x.\bar{b}),[l_i=\iota_i]^{i\in I})}(\mathrm{e{\cdot}upd_i})$$

By Lemma 23.(upd), there exists $[l_j{:}B_j,\ldots] \equiv B$ such that $type(a,B)$, $sub(B,A)$ and $x{:}B \vdash_{\mathsf{ND}} type(b,B_j)$. Since $eval(s,a,s',[l_i=\iota_i]^{i\in I})$, we can apply the inductive hypothesis, and deduce there exist C,Σ' such that:

$(a).\ res(\Sigma',[l_i=\iota_i]^{i\in I},C)$;
$(b).\ ext(\Sigma',\Sigma)$;
$(c).\ comp(\Sigma',s')$;
$(d).\ sub(C,B)$; that is, $C\equiv[l_j:B_j,\ldots]$.

Choose $A^+ := C$ and $\Sigma^+ := \Sigma'$.

By Lemma 23.$(bd{\cdot}weak)$ we obtain $x{:}C \vdash_{\mathsf{ND}} type(b,B_j)$; that is, using (a) and $j\in I$:

$$x{:}\Sigma_1^+(\iota_j) \vdash_{\mathsf{ND}} type(b,\Sigma_2^+(\iota_j)) \tag{5}$$

Next we derive $(\forall x,w,C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} res(\Sigma^+,w,C))$ from the fourth hypothesis of the theorem and Lemma 24.(iii). Then, because $closed(x) \vdash_{\mathsf{ND}} wrap(b,\bar{b})$ and (5), by Lemma 27.(i):

$$\Gamma, x{:}\Sigma_1^+(\iota_j) \vdash_{\mathsf{ND}} type_b(\Sigma^+,\bar{b},\Sigma_2^+(\iota_j)) \tag{6}$$

Since (c) we apply the Lemma 27.(ii), deriving $comp(\Sigma^+,(s'.\iota_j{\leftarrow}\lambda x.\bar{b}))$, and conclude by transitivity of subtyping.

($\mathrm{e{\cdot}clone}$). By hypothesis $type(clone(a),A)$ and:

$$\frac{eval(s,a,s',[l_i=\iota_i]^{i\in I}) \quad \forall i\in I : \iota_i' \notin \mathsf{Dom}(s')}{eval(s,clone(a),(s',\iota_i'{\mapsto}s'(\iota_i)^{i\in I}),[l_i=\iota_i']^{i\in I})}(\mathrm{e{\cdot}clone})$$

By Lemma 23.$(clone)$, there exists B such that $type(a,B)$ and $sub(B,A)$. Since $eval(s,a,s',[l_i=\iota_i]^{i\in I})$, we can apply the inductive hypothesis, thus deducing there exist C,Σ' such that:

$(a).\ res(\Sigma',[l_i=\iota_i]^{i\in I},C)$;
$(b).\ ext(\Sigma',\Sigma)$;
$(c).\ comp(\Sigma',s')$;
$(d).\ sub(C,B)$.

Choose $A^+ := C$ and $\Sigma^+ := \Sigma', (\iota_i'{\mapsto}\Sigma'(\iota_i))^{i\in I}$.

We deduce $ext(\Sigma^+,\Sigma)$ and $sub(A^+,A)$ by transitivity of ext and subtyping. Then we have $res(\Sigma^+,[l_i=\iota_i']^{i\in I},A^+)$ from (a) and Lemma 28.(i), and $comp(\Sigma^+,(s',\iota_i'{\mapsto}s'(\iota_i)^{i\in I}))$ using (c) and Lemma 28.(ii).

(e·let). By hypothesis $type(let(a, \lambda x.b), A)$ and:

$$\frac{eval(s, a, s', v) \quad \begin{array}{c}(x \mapsto v)\\ \vdots \\ eval(s', b, s'', v')\end{array}}{eval(s, let(a, \lambda x.b), s'', v')}(\text{e·let})$$

From Lemma 23.(let), there exist B, C such that $type(a, C)$, and $x{:}C \vdash_{\mathsf{ND}} type(b, B)$, and $sub(B, A)$. Since $eval(s, a, s', v)$, by inductive hypothesis there exist D, Σ' such that:

(a). $res(\Sigma', v, D)$;
(b). $ext(\Sigma', \Sigma)$;
(c). $comp(\Sigma', s')$;
(d). $sub(D, C)$.

Since $x{:}C \vdash_{\mathsf{ND}} type(b, B)$ and (d), we use Lemma 23.$(bd{\cdot}weak)$ for deriving $x{:}D \vdash_{\mathsf{ND}} type(b, B)$. Next we deduce $(\forall x, w, C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} res(\Sigma', w, C))$ from (a) and Lemma 24.(iii). Then, because $x{\mapsto}v \vdash_{\mathsf{ND}} eval(s', b, s'', v')$, we apply again the induction hypothesis, thus obtaining E, Σ'' such that $res(\Sigma'', v', E)$ and $ext(\Sigma'', \Sigma')$ and $comp(\Sigma'', s'')$ and $sub(E, B)$.

Choose $A^+ := E$, $\Sigma^+ := \Sigma''$, and conclude by transitivity of ext and subtyping.

(e·ground). By hypothesis $type_b(\Sigma, ground(a), A)$ and:

$$\frac{eval(s, a, s', v)}{eval_b(s, ground(a), s', v)}(\text{e·ground})$$

The assertion $type_b(\Sigma, ground(a), A)$ has to be derived by means of the rule (t·ground), namely from $type(a, A)$: therefore, by induction, there exist A^+, Σ^+ such that $res(\Sigma^+, v, A^+)$ and $ext(\Sigma^+, \Sigma)$ and $comp(\Sigma^+, s')$ and $sub(A^+, A)$.

(e·bind). By hypothesis $type_b(\Sigma, bind(v, \lambda y.\overline{b}), A)$ and:

$$\frac{\begin{array}{c}(y \mapsto v)\\ \vdots \\ eval_b(s, \overline{b}, s', v')\end{array}}{eval_b(s, bind(v, \lambda y.\overline{b}), s', v')}(\text{e·bind})$$

The assertion $type_b(\Sigma, bind(v, \lambda y.\overline{b}), A)$ has to be derived via (t·bind), so there exists B such that $res(\Sigma, v, B)$ and $y{:}B \vdash_{\mathsf{ND}} type_b(\Sigma, \overline{b}, A)$. Therefore, by mutual induction, there exist A^+, Σ^+ such that $res(s', v', A^+)$ and $ext(\Sigma^+, \Sigma)$ and $comp(\Sigma^+, s')$ and $sub(A^+, A)$. □

A.2. SUBJECT REDUCTION WITH COINDUCTIVE RESULT TYPING (FUN_ς)

LEMMA 30 (Coinductive result typing).

$$
\begin{array}{lll}
(i) & : & cores(s,v,A) \wedge eval(s,a,s',v') \;\Rightarrow \\
 & & cores(s',v,A) \\
 & & \\
(ii) & : & cores(s,v,A) \;\Rightarrow\; cores((s,t),v,A) \\
 & & \\
(iii) & : & x{:}A \vdash_{\mathsf{ND}} type(b,B) \wedge closed(x) \vdash_{\mathsf{ND}} wrap(b,\overline{b}) \wedge \\
 & & (\forall x,w,C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} cores(s,w,C)) \;\Rightarrow \\
 & & x{:}A \vdash_{\mathsf{ND}} cotype_b((s,\iota{\mapsto}\lambda x.\overline{b}),\overline{b},B)
\end{array}
$$

Proof. (i) By structural induction on the derivation of $eval(s,a,s',v')$. (ii) By structural coinduction. (iii) By structural induction on the derivation of $closed(x) \vdash_{\mathsf{ND}} wrap(b,\overline{b})$, and point (ii). □

LEMMA 31 (Objects).

$$
\begin{array}{l}
type([l_i{=}\varsigma(x_i)b_i]^{i\in I}, [l_i{:}B_i]^{i\in I}) \wedge closed(x_i) \vdash_{\mathsf{ND}} wrap(b_i,\overline{b}_i)^{i\in I} \wedge \\
(\forall x,w,C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} cores(s,w,C)) \;\Rightarrow \\
cores((s,\iota_i{\mapsto}\lambda x_i.\overline{b}_i)^{i\in I}, [l_i{=}\iota_i]^{i\in I}, [l_i{:}B_i]^{i\in I})
\end{array}
$$

Proof. By induction on the object $[l_i{=}\varsigma(x_i)b_i]^{i\in I}$, and Lemmas 30.$(ii)$ and 30.(iii). □

LEMMA 32 (Method invocation).

$$
\begin{array}{l}
cores(s,[l_j{=}\iota_j,\ldots],[l_j{:}B_j,\ldots]) \wedge s(\iota_j) = \lambda x.\overline{b} \;\Rightarrow \\
x{:}[l_j{:}B_j,\ldots] \vdash_{\mathsf{ND}} cotype_b(s,\overline{b},B_j)
\end{array}
$$

Proof. By inspection on the rule $(\mathrm{t}{\cdot}\mathrm{cores})$. □

LEMMA 33 (Functional method update).

$$
\begin{array}{l}
cores(s,[l_i{=}\iota_i]^{i\in I},[l_j{:}B_j,\ldots]) \wedge (j \in I) \\
x{:}[l_j{:}B_j,\ldots] \vdash_{\mathsf{ND}} cotype_b((s,\iota{\mapsto}\lambda x.\overline{b}),\overline{b},B_j) \wedge \\
(\forall x,w,C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} cores(s,w,C)) \;\Rightarrow \\
cores((s,\iota{\mapsto}\lambda x.\overline{b}),[l_i{=}\iota_i, l_j{=}\iota]^{i\in I\setminus\{j\}},[l_j{:}B_j,\ldots])
\end{array}
$$

Proof. By induction on the result $[l_i{=}\iota_i]^{i\in I}$, and Lemma 30.(ii). □

THEOREM 34 (Subject Reduction with coinductive result typing).

$$
\begin{array}{l}
\Gamma \vdash_{\mathsf{ND}} type(a,A) \;\wedge\; \Gamma \vdash_{\mathsf{ND}} eval(s,a,t,v) \wedge \\
(\forall x,w,C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} cores(s,w,C)) \;\Rightarrow \\
\exists A^+{:}TType : \Gamma \vdash_{\mathsf{ND}} cores(t,v,A^+) \;\wedge\; \Gamma \vdash_{\mathsf{ND}} sub(A^+\; A)
\end{array}
$$

Proof. By structural induction on the derivation of $\Gamma \vdash_{\mathsf{ND}} eval(s, a, t, v)$. The rules $(e{\cdot}call)$ and $(e{\cdot}bind)$ require a *mutual* structural induction argument, namely a stronger induction schema valid also for the predicate $eval_b$, which is the counterpart of $eval$ for closures.

(e·var). By hypothesis $type(x, A)$ and:

$$\frac{x{\mapsto}v}{eval(s, x, s, v)}(\text{e·var})$$

From Lemma 23.(var), there exists B such that $x{:}B \in \Gamma$ and $sub(B, A)$. Choose $A^+ := B$.

Since $x{\mapsto}v \in \Gamma$, by the third hypothesis of the theorem we can derive $cores(s, v, A^+)$, thus concluding.

(e·obj). By hypothesis $type([l_i = \varsigma(x_i)b_i]^{i\in I}, A)$ and:

$$\frac{\forall i \in I : \iota_i \notin \mathsf{Dom}(s) \quad \begin{array}{c}(closed(x_i))\\ \vdots \\ wrap(b_i, \overline{b}_i)\end{array}}{eval(s, [l_i = \varsigma(x_i)b_i]^{i\in I}, (s, \iota_i{\mapsto}\lambda x_i.\overline{b}_i)^{i\in I}, [l_i = \iota_i]^{i\in I})}(\text{e·obj})$$

By Lemma 23.(obj), there exists $[l_i{:}B_i]^{i\in I}$ such that:

$$type([l_i = \varsigma(x_i)b_i]^{i\in I}, [l_i{:}B_i]^{i\in I}) \tag{7}$$

and $sub([l_i{:}B_i]^{i\in I}, A)$. Choose $A^+ := [l_i{:}B_i]^{i\in I}$.

Since $closed(x_i) \vdash_{\mathsf{ND}} wrap(b_i, \overline{b}_i)\ \forall i \in I$ and (7), we apply Lemma 31, and deduce $cores((s, \iota_i{\mapsto}\lambda x_i.\overline{b}_i)^{i\in I}, [l_i = \iota_i]^{i\in I}, A^+)$.

(e·call). By hypothesis $type(a.l_j, A)$ and:

$$\frac{eval(s, a, s', [l_i = \iota_i]^{i\in I}) \quad s'(\iota_j) = \lambda x.\overline{b}_j \quad \begin{array}{c}(x{\mapsto}[l_i = \iota_i]^{i\in I})\\ \vdots \\ eval_b(s', \overline{b}_j, s'', v)\end{array} \quad j \in I}{eval(s, a.l_j, s'', v)}(\text{e·call})$$

By Lemma 23.$(call)$, there exists $[l_j{:}B_j, \ldots]$ such that $type(a, [l_j{:}B_j, \ldots])$ and $sub(B_j, A)$. Since $eval(s, a, s', [l_i = \iota_i]^{i\in I})$, by inductive hypothesis there exists C such that:

(a) $cores(s', [l_i = \iota_i]^{i\in I}, C)$;

(b) $sub(C, [l_j{:}B_j, \ldots])$.

Among the premises of the rule (e·call), we have $j \in I$, $s'(\iota_j) = \lambda x.\overline{b}_j$, and:

$$x{\mapsto}[l_i = \iota_i]^{i\in I} \vdash_{\mathsf{ND}} eval_b(s', \overline{b}_j, s'', v) \tag{8}$$

Moreover, it is $C \equiv [l_j{:}B_j, \ldots]$ from (b); therefore, using (a) and Lemma 32:

$$x{:}C \vdash_{\mathsf{ND}} cotype_b(s', \overline{b}_j, B_j) \tag{9}$$

We deduce $(\forall x, w, C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} cores(s', w, C))$ from the third hypothesis of the theorem and Lemma 30.(i). Then, since (8) and (9), we apply the mutual induction hypothesis, deducing there exists A^+ such that $cores(s'', v, A^+)$ and $sub(A^+, B_j)$. We finish by transitivity of subtyping.

$(\mathrm{e}{\cdot}\mathrm{upd_f})$. By hypothesis $type(a.l{\leftarrow}\varsigma(x)b, A)$ and:

$$\frac{\begin{array}{c}(closed(x))\\ \vdots\\ eval(s, a, s', [l_i = \iota_i]^{i\in I}) \quad wrap(b, \overline{b}) \quad \iota'_j \notin \mathsf{Dom}(s') \quad (j \in I)\end{array}}{eval(s, a.l{\leftarrow}\varsigma(x)b, (s', \iota'_j{\mapsto}\lambda x.\overline{b}), [l_i = \iota_i, l_j = \iota'_j]^{i\in I\setminus\{j\}})}(\mathrm{e}{\cdot}\mathrm{upd_f})$$

By Lemma 23.(upd), there exists $[l_j{:}B_j, \ldots] \equiv B$ such that $type(a, B)$, $sub(B, A)$ and $x{:}B \vdash_{\mathsf{ND}} type(b, B_j)$. Since $eval(s, a, s', [l_i = \iota_i]^{i\in I})$, we can apply the inductive hypothesis, deducing there exist C such that:

(a). $cores(s', [l_i = \iota_i]^{i\in I}, C)$;

(b). $sub(C, B)$; that is, $C \equiv [l_j : B_j, \ldots]$.

Choose $A^+ := C$.

By Lemma 23.$(bd{\cdot}weak)$, we obtain $x{:}A^+ \vdash_{\mathsf{ND}} type(b, B_j)$; then, exploting $closed(x) \vdash_{\mathsf{ND}} wrap(b, \overline{b})$ and Lemma 30.(iii), we deduce:

$$x{:}A^+ \vdash_{\mathsf{ND}} cotype_b((s', \iota'_j{\mapsto}\lambda x.\overline{b}), \overline{b}, B_j) \tag{10}$$

Next we derive $(\forall x, w, C : (x{\mapsto}w, x{:}C \in \Gamma) \Rightarrow \Gamma \vdash_{\mathsf{ND}} cores(s', w, C))$ from the third hypothesis of the theorem and Lemma 30.(i). Then, from (a), (10), $j \in I$ and Lemma 33, we have:

$$cores((s', \iota'_j{\mapsto}\lambda x.\overline{b}), [l_i = \iota_i, l_j = \iota'_j]^{i\in I\setminus\{j\}}, A^+)$$

and conclude by transitivity of subtyping.

$(\mathrm{e}{\cdot}\mathrm{ground})$. By hypothesis $cotype_b(s, ground(a), A)$ and:

$$\frac{eval(s, a, s', v)}{eval_b(s, ground(a), s', v)}(\mathrm{e}{\cdot}\mathrm{ground})$$

The assertion $cotype_b(s, ground(a), A)$ has to be derived by means of the rule $(\mathrm{t}{\cdot}\mathrm{coground})$, namely from $type(a, A)$: therefore, by induction, there exists A^+ such that $cores(s', v, A^+)$ and $sub(A^+, A)$.

$(\mathrm{e}{\cdot}\mathrm{bind})$. By hypothesis $cotype_b(s, bind(v, \lambda y.\overline{b}), A)$ and:

$$\frac{\begin{array}{c}(y{\mapsto}v)\\ \vdots\\ eval_b(s, \overline{b}, s', v')\end{array}}{eval_b(s, bind(v, \lambda y.\overline{b}), s', v')}(\mathrm{e}{\cdot}\mathrm{bind})$$

The assertion $cotype_b(s, bind(v, \lambda y.\overline{b}), A)$ must be derived via $(\mathrm{t}{\cdot}\mathrm{cobind})$, so exists B such that $cores(s, v, B)$ and $y{:}B \vdash_{\mathsf{ND}} cotype_b(s, \overline{b}, A)$. By mutual induction, there exists A^+ such that $cores(s', v', A^+)$ and $sub(A^+, A)$. $\square$