Imperative Objects and Mobile Processes

DRAFT

Josva Kleist* Davide Sangiorgi†

October, 1997

Abstract

An interpretation of Abadi and Cardelli's first-order *Imperative Object Calculus* into a typed π-calculus is presented. The interpretation validates the subtyping relation and the typing judgements of the Object Calculus, and is computationally adequate. The proof of computational adequacy makes use of (a π-calculus version) of *ready simulation*, and of a *factorisation* of the interpretation into a functional part and a very simple imperative part. The interpretation can be used to compare and contrast the Imperative and the *Functional* Object Calculi, and to prove properties about them, within a unified framework.

1 Introduction

In their book [1], Abadi and Cardelli present and investigate a *Functional* and an *Imperative Object Calculus*, and type systems for them. These calculi can express, as primitive or derived forms, various major object-oriented idioms; they have simple but interesting typing and subtyping rules. The syntactic simplicity of the calculi, and their clear object-oriented flavour, makes them an important basis for understanding object-oriented languages. All Object Calculi are sequential.

In this paper we study the interpretation of the (first order) Imperative Object Calculus (IOC) into a *typed π-calculus*. Our main motivations are:

1. There is a general lack of mathematical techniques for giving the semantics to, and proving properties about, object-oriented languages , especially the imperative ones. (For instance, it seems difficult to come up with reasonable notions of bisimulation for IOC. This contrasts with the Functional Object Calculus (FOC), for which one such notion has been developed.)

---

*BRICS, Department of Computer Science, Aalborg University, Denmark. Email: `kleist@cs.auc.dk`

†INRIA Sophia-Antipolis, France. Email: `davide.sangiorgi@sophia.inria.fr`

However, most “real world” programming languages are imperative. Usually, objects encapsulate a state, which can be manipulated by activating the methods of the object.

The π-calculus has a rich algebraic theory and a high expressive power. Its emphasis on the notions of name and of mobility makes it appealing for describing objects and their local states.

2. IOC is an interesting core object-oriented language, because it is small and yet very expressive; for instance classes and functions, as well as the Functional Object Calculus (FOC), can be encoded in it. A study of IOC can provide a solid basis for investigating more complex languages, that may include also, for instance, constructs for distribution and concurrency.

3. We wish to understand what are objects from a π-calculus (and more generally, a process calculus) point of view.

The only work on behavioural equivalences for IOC that we are aware of is Gordon, Hankin and Lassen's [7]. In this work, however, IOC is *untyped*. Gordon, Hankin and Lassen study contextual equivalence for this untyped IOC, prove that it coincides with a variant of Mason and Talcott's CIU equivalence [15], and use the latter to validate some basic laws of the calculus.

More work exists on the semantics of the Functional Object Calculus FOC. Typed contextual equivalence and applicative bisimulation for FOC have been examined by Gordon and Rees [8], who show that these two notions of equivalence coincide. They also show that Abadi and Cardelli's equational theory for FOC [1] is sound w.r.t. operational equivalence. Abadi and Cardelli [1, Chapter 14] shows that the equational theory for FOC is also sound with respect their denotational semantics of FOC. Aceto et al. [3] show that the denotational semantics is sound but not fully abstract w.r.t. operational equivalence. Extending the above-mentioned techniques, based on applicative bisimulation and denotational models, to the Imperative Object Calculus IOC appears rather hard.

An interpretation of IOC into a form of imperative polymorph λ-calculus with subtyping, recursive types and records has been found by Abadi, Cardelli and Viswanathan [2]. This interpretation has been used to validate the subtyping and typing judgements of IOC. However, it would be difficult to prove behavioural properties of IOC from this interpretation, because very little is known of the theory of the target imperative λ-calculus.

Some previous studies of encodings of imperative or OOLs into process calculi, namely [16, Chapter 8], [27], [11], [29, 14], and [24, 10], are an important basis for our work. We briefly comment on the differences. Milner [16, Chapter 8] showed how to translate a small imperative language into CCS. Vaandrager [27], Jones [11] and Walker, Liu and Philippou [29, 14, 20] have gone further, by translating parallel object-oriented languages derived from the POOL family. Walker, Liu and Philippou have also used the encodings for proving the validity of certain program transformations on the source languages. The main limitation of these works is that they do not show how to handle *typed* object-oriented languages — the source languages have rather simple type systems and the translations do not act on types. Dealing with types is important when the type system of the object-oriented language contains non-trivial features like subtyping and polymorphism, otherwise many useful program equalities are lost and the semantics cannot be used to validate the typing rules of the language. By contrast, types play a central role in our interpretation. For this reason, our interpretation of objects is different from those in the above-mentioned works. In [10, 24], interpretations of, respectively, untyped and typed FOC into the π-calculus are given. No use is made of the π-calculus interpretation for for validating behavioural properties of the source Object Calculus.

The syntax of FOC and IOC are similar, but their operational semantics are very different (for instance, the operational semantics of IOC makes an extensive use of stores and stacks, not needed for FOC because it is functional). Remarkably, despite these differences in the operational semantics, the π-calculus interpretation of IOC in this paper can be derived with a simple change from that of FOC in [24]. As a consequence we can use the π-calculus interpretations to compare and contrast the Imperative and Functional Object Calculi — for instance, their discriminating power — and to prove properties about them, within a single framework.

The type language we use for the π-calculus is taken from [21, 24]. In these type systems, as well as other type systems for the π-calculus like [12, 28, 9], types are assigned to names. The types in [21] show the arity and the directionality of a name and, recursively, of the names carried by that name; the difference in [24] is *variant types* in place of tupling. All values are "simple", in the sense that they are built out of names only, and cannot contain, for instance, process expressions.

The main technical contents of this paper are the following. We give a translation of both IOC terms and IOC types and type environments into the typed π-calculus. We then provide correctness proofs. Precisely, we prove that (1) the translation validates the subtyping judgements of IOC, that is, A is a subtype of B iff the translation of A is a subtype of the translation of B; (2) a IOC type judgement $E \vdash a : A$, asserting that object a has type A under the type assumptions E, is true iff its π-calculus translation is true, and (3) a well-typed IOC term reduces to a value iff its π-calculus interpretation does so (computational adequacy). From these results and the compositionality of the encoding, as a corollary we get the soundness of the translation w.r.t. behavioural equivalences like Morris-style contextual equivalence or barbed congruence [19]. Soundness assures us that the equalities on IOC terms provable from the interpretation are operationally valid on IOC. As for the translations of λ-calculi into π-calculus, the opposite implication fails.

The IOC language we interpret into the π-calculus has a first-order type system. The interpretation can be easily extended to accommodate other features, like recursive types, variant tags, and polymorphism.

Technically, the hardest part of our work is the proof of computational adequacy. The proof is split into two parts. First we factorize our encoding into a functional part, where processes are "stateless", and a very simple imperative part, where processes have a state. This factorisation is useful because: (1) it allows us take full advantage of various π-calculus proof techniques for functional processes; (2) it shows what – we believe – are the simplest non-functional π-calculus processes that are needed for translating the imperative features of IOC. This factorised encoding is less compact than the original one; but it allows us to establish a close correspondence with Abadi-Cardelli's operational semantics of IOC.

The second part of the proof of computational adequacy is to establish a relation between the original encoding and the factorised one. To this end, we could not use known behavioural equivalences of the π-calculus (for instance, known bisimilarities were too strong; trace equivalence too weak to yield the desired property). We solved the problem by adapting (a weak version of) the notion of *ready simulation* [4, 13] to the π-calculus. Roughly, ready simulation was introduced in CCS-like languages as the least congruence contained in trace inclusion induced by certain classes of operators. To our knowledge, our application of ready simulation to derive properties of processes is novel.

Having established the correctness of our interpretation, we give some examples of how the theory of the π-calculus can be used to reason about IOC. The advantage of using π-calculus for the proofs is that we can take advantage of the already available theory, including its algebraic laws and its co-inductive proof techniques. Here, we use the π-calculus to validate some basic equational theory for IOC. Something interesting about the coinductive π-calculus proofs is that non-trivial equalities can be proved using finitary relations (this is more rare with CIU equivalence on untyped IOC [7] or applicative bisimulation on FOC [8] because the definition, or the transition system, on which they are based, contains an infinite quantification on contexts, or terms, of the language).

Among the laws we prove is (EQ SUB OBJECT). This law allows us to eliminate methods of objects that are not visible in the type assigned to the object. We are not aware of proposals of this, or similar laws, for IOC. In Abadi-Cardelli's book, the analogous of law (EQ SUB OBJECT) is at the heart of the equational theory of FOC, but no equational theory for IOC is proposed (in the book or, as far as we know, elsewhere). Strikingly, we can prove (EQ SUB OBJECT) using a bisimulation relation consisting of just two elements. Essentially the same proof can be used for (EQ SUB OBJECT) on FOC.

In this short version of the paper, for lack of space most of the proofs are omitted.

2 The Imperative Object Calculus

In this section we present the (first-order) Imperative Object Calculus (IOC) from [1]. Grammar, operational and typing rules are reported in Appendix A.

Syntax. An object $[l_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}]$ consists of a collection of named methods $l_i = \varsigma(x_i{:}A)b_i$ for distinct names l_i, and where x_j's are the self parameters. The letter ς (sigma) is a binder for the self variable in a method; $\varsigma(x{:}A)b$ is a method that when activated evaluates its body b with x bound to the enclosing object. A method activation $a.l$ results in the activation of the method bound to l in a; this method is evaluated with a as argument for self. Instantiating this parameter at call time results in what is named *late binding* — the self of a method can change dynamically at runtime. A method update $a.l{\Leftarrow}\varsigma(x{:}A)b$ replaces the method named l in the object a with $\varsigma(x{:}A)b$ and evaluates to the modified object. Cloning creates a copy of the original object. A $\mathsf{let}\ x{:}A = a\ \mathsf{in}\ b$ expression first evaluates the let-part, binding the result to x, then the in-part is evaluated with the variable x in scope. Sequential evaluation of objects $a; b$ can be defined thus: $\mathsf{let}\ x{:}A = a\ \mathsf{in}\ b$ for some $x \notin \mathsf{fv}(b)$.

Operational semantics. Object terms are evaluated w.r.t. a global store σ. If the evaluation of an object terminates, the object reduces to a *value* $[l_i{=}\iota_i\ ^{i\in 1..n}]$ and an updated store. The store σ contains closures; a closure is the pair $\langle\varsigma(x)b, \mathcal{S}\rangle$ of a method body $\varsigma(x)b$ together with a local stack. A stack $\mathcal{S}$ maps variables to values. For any mapping f we let $\mathsf{dom}(f)$ denote the domain of f. A stack $\mathcal{S}$ is well-formed w.r.t. a store σ, written $\sigma \cdot \mathcal{S} \vdash \diamond$ if for all $x \in \mathsf{dom}(\mathcal{S})$ $\mathcal{S}(x) = [l_i{=}\iota_i\ ^{i\in 1..n}]$ all ι_i is defined in σ. A store σ is well-formed, written $\sigma \vdash \diamond$ if the stacks in all closures in the store are well-formed w.r.t. the store.

If σ is a store, then we let $\sigma, \iota \mapsto \langle\varsigma(x)b, \mathcal{S}\rangle$ denote the extension of σ with the new entry $\langle\varsigma(x)b, \mathcal{S}\rangle$ on the *new* location ι, assuming that $\sigma \cdot \mathcal{S} \vdash \diamond$. We write $\sigma[\iota \mapsto \langle\varsigma(x)b, s\rangle]$ for the update of location ι of store σ, assuming that $\iota \in \mathsf{dom}(\sigma)$ and again $\sigma \cdot \mathcal{S} \vdash \diamond$.

The operational semantics is untyped; type annotations are simply removed when evaluating terms. We write $a{\Downarrow}v \cdot \sigma$ ("a converge to the value v and store σ") if we can deduce $\emptyset \cdot \emptyset \vdash a \leadsto v \cdot \sigma$ and $a{\Downarrow}$ iff $a{\Downarrow}v \cdot \sigma$ for some v and σ. If there is no v and σ such that $a{\Downarrow}v \cdot \sigma$ we write $a{\Uparrow}$ ("a diverges").

Type system. There are two forms of judgments for IOC: Type judgments and subtyping judgments. *Type judgments* are of the form $E \vdash a{:}A$ and state that the object a has type A under the assumptions in E, where E describes typing assumptions for free self variables. If E is empty we shall sometimes just write $a{:}A$ instead of $\emptyset \vdash a{:}A$. *Subtype judgments* $A <: B$ state that the type A is a subtype of B. Subtyping allows an object to be replaced by another with additional methods, but the common methods must have the same type. This invariance is necessary for the soundness of the reduction rules, i.e. avoiding requests on methods which do not exist.

3 A typed mobile calculus

In this section we present the typed π-calculus on which we shall interpret IOC.

Syntax. The syntax of the typed π-calculus is given in Table 1. The process constructs are those of the monadic π-calculus [18] with matching replaced by a case construct. The latter can be thought of as a more disciplined form of matching, in which all tests on a given name are localised to a single place. The construct wrong stands for a process in which a run-time type error has occurred — i.e., for instance a communication in which the variant tag or the arity of the transmitted value was unexpected by its recipient. A soundness theorem guarantees that a well-typed process expression cannot reduce to an expression containing wrong.

Names

$p, q, r \dots x, y, z$

Variant Tags

ℓ, l, h

Types

T	$::=$	$\mu(X)T$	recursive type
	$\mid$	X	type variable
	$\mid$	$[\,\ell_1\_T_1..\ell_n\_T_n\,]$	variant type
	$\mid$	$\langle T_1 \dots T_n \rangle$	tuple type
	$\mid$	T^I	channel type

I/O Tags

I	$::=$	r	input only
	$\mid$	w	output only
	$\mid$	b	either

Values

v	$::=$	x	name
	$\mid$	$\langle v_1..v_n \rangle$	tuple value
	$\mid$	ℓ_v	variant value

Processes

P	$::=$	$\mathbf{0}$	nil process
	$\mid$	$P \mid P$	parallel
	$\mid$	$(\nu x{:}T)P$	restriction
	$\mid$	$p(x).P$	input
	$\mid$	$\overline{p}v.P$	output
	$\mid$	$!P$	replication
	$\mid$	$\mathsf{let}\ x_1..x_n = v\ \mathsf{in}\ P$	tuple destructor
	$\mid$	$\mathtt{case}\ v\ \mathtt{of}\ [_{j \in 1..n}\ \ell_j\_(x_j) \triangleright P_j]$	case
	$\mid$	wrong	error

where:

- In a recursive type $\mu(X)T$, variable X must be guarded in T, i.e., occur underneath a I/O-tag or underneath a variant tag;
- in the case statement, the tags ℓ_i $(i \in 1..n)$ are pairwise distinct.

Table 1: The syntax of the typed π-calculus

Operational semantics. For the semantics of the π-calculus we adopt a labelled transition system. The advantage of a labeled semantics, compared to a reduction semantics [17, 24], is that it easily allows us to define labeled forms of bisimulation. Process transitions are of the form $P \xrightarrow{\mu} P'$, where μ is given by the following syntax:

$$\mu ::= (\nu\tilde{n}{:}\tilde{T})\bar{p}v \mid pv \mid \tau \mid \mathsf{wrong}$$

The label $(\nu\tilde{n}{:}\tilde{T})\bar{p}v$ denotes the output of the value v on the name p. The restriction $(\nu\tilde{n}{:}\tilde{T})$ where $\tilde{n}$ must be a subset of the names in v indicates that the names $\tilde{n}$ are bound names having types $\tilde{T}$. The label pv denotes the input of the value v over the name p. The action τ denotes an internal action. Finally, wrong denotes a run-time error. The rules for the operational semantics are standard rules of the π-calculus; for instance the rules for input, and the communication rule are:

$$\frac{-}{p(x).P \xrightarrow{pv} P\{v/x\}} \qquad \frac{P \xrightarrow{(\nu\tilde{n}:\tilde{T})\bar{p}v} P' \quad Q \xrightarrow{pv} Q'}{P \mid Q \xrightarrow{\tau} (\nu\tilde{n}:\tilde{T})(P' \mid Q')}\ \tilde{n} \cap \mathsf{fn}(Q) = \emptyset$$

The new, but expected, rules are those for `let` and `case`, in which run-time errors may be generated. For `case` we have (`let` is similar):

$$\frac{\ell_j \in \{\ell_1 \ldots \ell_n\}}{\mathsf{case}\ \ell_j\_v\ \mathsf{of}\ [\,\ell_1\_(x_1) \rhd P_1; \ldots; \ell_n\_(x_n) \rhd P_n\,] \xrightarrow{\tau} P_j\{v/x\}}$$

$$\frac{v \neq \ell_j\_v' \text{ or } \ell_j \notin \{\ell_1 \ldots \ell_n\}}{\mathsf{case}\ v\ \mathsf{of}\ [\,\ell_1\_(x_1) \rhd P_1; \ldots; \ell_n\_(x_n) \rhd P_n\,] \xrightarrow{\mathsf{wrong}} \mathsf{wrong}}$$

We write $P \xrightarrow{\mu}_d Q$ if $P \xrightarrow{\mu} Q$ is the only transitions that P can perform. And as usual we let $\stackrel{\mu}{\Longrightarrow}$ denote weak transitions, and $P \Longrightarrow P'$ means "$P \stackrel{\tau}{\Longrightarrow} P'$ or $P = P'$".

Typing and subtyping. We recall that I/O annotations [21] are to separate the capabilities of reading and writing on a channel (we use "read" and "write" as synonymous for "input" and "output", respectively). For instance, a type $p : \langle S^{\mathsf{r}}\, T^{\mathsf{w}} \rangle^{\mathsf{b}}$ (for appropriate type expressions S and T) says that name p can be used *both* to read and to write and that any message at p carries a pair of names; moreover, the first component of the pair can be used by the recipient *only to read*, the second *only to write*.

Subtyping judgements, shown in Table 2, are of the form $\Sigma \vdash S <: T$, where Σ represents the subtyping assumptions. We often write $S <: T$ when the subtyping assumptions are empty. Note that type annotation r (an input capability) gives covariance, w (an output capability) gives contravariance, and b (both capabilities) gives invariance. A *type environment* Γ a finite assignment of types to names.

A typing judgement $\Gamma \vdash P$ asserts that process P is well-typed in Γ, and $\Gamma \vdash v : T$ (Table 2) that value v has type T in Γ. There is one typing rule for each process construct except wrong. The interesting rules are those for input and output prefixes and for case. In the rules for input and output prefixes, the subject of the prefix is checked to possess the appropriate input or output capability in the type environment. A subject reduction theorem guarantees us that if P is well-typed and $P \Longrightarrow Q$, then Q does not contain the process wrong.

Some derived constructs. In the translation we shall use: *Recursive definitions*, $A(\tilde{x}) \stackrel{\mathrm{def}}{=} P$ which can be defined the standard way from replication (c.f. [17]); *polyadic inputs* $a(x_1 \ldots x_n).P$, defined as $a(y).\mathsf{let}\ \langle x_1 \ldots x_n \rangle = y\ \mathsf{in}\ P$ for $y \notin \mathsf{fn}(P)$; *variant inputs*, like $p\big[_{j\in 1..n}\ \ell_j\_\big[_{i\in 1..m}\ \ell_{i,j}\_(\tilde{x}_{i,j}) \rhd P_{i,j}\big]\big]$. The last abbreviation allows us to go down two levels into the structure of a variant value received in an input at p; in fact, this term interacts with output particles

Subtyping rules:

$$\frac{\Sigma \vdash S <: T \quad \Sigma \vdash T <: S}{\Sigma \vdash S^{\mathsf{b}} <: T^{\mathsf{b}}} \qquad \frac{I \in \{\mathsf{b}, \mathsf{r}\} \quad \Sigma \vdash S <: T}{\Sigma \vdash S^{I} <: T^{\mathsf{r}}}$$

$$\frac{I \in \{\mathsf{b}, \mathsf{w}\} \quad \Sigma \vdash T <: S}{\Sigma \vdash S^{I} <: T^{\mathsf{w}}} \qquad \frac{\Sigma \vdash S_i <: T_i \quad i \in 1..n}{\Sigma \vdash [\, \ell_1\_S_1..\ell_n\_S_n \,] <: [\, \ell_1\_T_1..\ell_{n+m}\_T_{n+m} \,]}$$

$$\frac{\Sigma \vdash S_i <: T_i \ \forall i \in 1..n}{\Sigma \vdash \langle S_1..S_n \rangle <: \langle T_1..T_n \rangle} \qquad \frac{-}{\Sigma, S <: T, \Sigma' \vdash S <: T}$$

$$\frac{\Sigma, \mu(X)S <: T \vdash S\{{}^{\mu(X)S}/_{X}\} <: T}{\Sigma \vdash \mu(X)S <: T} \qquad \frac{\Sigma, S <: \mu(X)T \vdash S <: T\{{}^{\mu(X)T}/_{X}\}}{\Sigma \vdash S <: \mu(X)T}$$

Process typing:

$$\frac{-}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P|Q} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P} \qquad \frac{\Gamma, x{:}S^{I} \vdash P}{\Gamma \vdash (\nu x{:}S^{I})P}$$

$$\frac{\Gamma \vdash p : S^{\mathsf{w}} \quad \Gamma \vdash w : S \quad \Gamma \vdash P}{\Gamma \vdash \overline{p}w.P} \qquad \frac{\Gamma \vdash v : \langle T_1..T_n \rangle \quad \Gamma, x_1{:}T_1, \ldots x_n{:}T_n \vdash P}{\Gamma \vdash \mathsf{let}\ x_1..x_n = v\ \mathsf{in}\ P}$$

$$\frac{\Gamma \vdash p : S^{\mathsf{r}} \quad \Gamma, x{:}S \vdash P}{\Gamma \vdash p(x).P} \qquad \frac{\Gamma \vdash v : [\, \ell_1\_T_1..\ell_n\_T_n \,] \quad \text{for each } i, \ \ \Gamma, x_i{:}T_i \vdash P_i}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ [\, \ell_1\_(x_1) \triangleright P_1; \ldots ; \ell_n\_(x_n) \triangleright P_n \,]}$$

Value typing:

$$\frac{\Gamma(p) = T}{\Gamma \vdash p : T} \qquad \frac{\Gamma \vdash v : S \quad S <: T}{\Gamma \vdash v : T} \qquad \frac{\Gamma \vdash v : T}{\Gamma \vdash \ell\_v : [\ell\_T]} \qquad \frac{\Gamma \vdash v_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash \langle x_1..x_n \rangle : \langle T_1..T_n \rangle}$$

Table 2: Subtyping and typing for the π-calculus

of the form $\overline{p}\ell_r\_\ell_{s,r}\_\tilde{w}$ (with $r \in 1..n$, $s \in 1..m$, and tuple $\tilde{w}$ of the same length as $\tilde{x}_{r,s}$) and, in doing so, it reduces to $P_{r,s}\{{}^{\tilde{w}}/_{\tilde{x}_{r,s}}\}$.

Barbed bisimulation and congruence. The behavioural equality we adopt for the π-calculus is *barbed congruence.* Barbed congruence is a bisimulation-based relation that has been used for a broad variety of calculi; its main advantage is that it can be uniformly defined on different calculi, for it requires of the calculus little more than a notion of reduction – the τ-step of the π-calculus.

Barbed congruence is defined as the congruence induced by *barbed bisimulation.* We write $P \Downarrow_p$ if P is *observable at* p, that is P can accept an input or an output communication with the environment along p; formally $P \Downarrow_p$ is true if $P \overset{\mu}{\Longrightarrow} P'$, for some P' and μ where μ is an input or output action at p.

Definition 1 (barbed bisimulation and congruence) Barbed bisimulation *is the largest symmetric relation* $\dot{\approx}$ *on processes s.t.* $P \dot{\approx} Q$ *implies:*

1. whenever $P \Longrightarrow P'$ then there exists Q' such that $Q \Longrightarrow Q'$ and $P' \stackrel{\bullet}{\approx} Q'$;

2. for each name p, $P \Downarrow_p$ iff $Q \Downarrow_p$.

Two processes P and Q are barbed congruent*, written $P \approx Q$, if $C[P] \stackrel{\bullet}{\approx} C[Q]$ for all contexts C.*

In an untyped calculus, no contraint is made on processes or contexts. In a typed setting, process compared should have the same type, and should only be tested in contexts which respect such a type. We write $P \approx_\Gamma Q$ if $\Gamma \vdash P, Q$ and $C[P] \stackrel{\bullet}{\approx} C[Q]$ holds, in all contexts that respect Γ. We omit the formal definitions of the typed relations.

Barbed congruence requires a quantification over all contexts. Therefore proving process equalities can be heavy. Against this, it is important to have powerful proof techniques. One such a technique consists in using *labeled bisimilarities* whose definition does not require context quantification. For instance, in the untyped π-calculus barbed congruence can be recovered using the well-known *early labeled bisimilarity.* Its typed version, *typed early labeled bisimulation* is studied by Boreale and Sangiorgi [5]. These techniques of labeled bisimilarities can be made more powerful by combining then with up-to techniques, like “up to parallel composition” and “up to injective substitutions”.

4 The interpretation

To understand the π-calculus interpretation, it may be helpful to see first an intermediate interpretation into the *Higher-Order π-calculus* (HOπ) [23], an extension of the π-calculus where arguments of communications and recursive definitions may be, besides names, also *abstractions*, i.e., parameterised processes. For the interpretation of IOC, we only need abstractions in recursive definitions. More precisely, we need certain parameters of recursive definitions to be functions from names to processes. An example of such a recursive definition is

$$K(f,p) \stackrel{\text{def}}{=} p(x).(f\langle p\rangle | K\langle f, x\rangle)$$

Here, f is a function parameter, and p a name parameter; $f\langle p\rangle$ is the process obtained by applying function f to name p. We write functions from names to processes using a lambda notation, like in $\lambda(x,y).P$. The interpretation into HOπ is shown in Table 3. We have omitted the type annotations.[1].

The translation $\{\!\![a]\!\!\}_p$ of an IOC term a is located at a channel p. When a is an object value, with methods l_i $(i \in 1..n)$, its translation is a process whose first action is to signal its valuehood by providing an access s to its value-core, which is a process of the form $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$. This process is ready to accept along the access name s requests of selection, update and cloning for the methods l_j.

---

[1] We are omitting type annotations for ease of reading; they would be however necessary to make the definition formal. Types are taken into account in the interpretations into the π-calculus in Table 4

$$
\begin{array}{rcl}
\{\![[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]]\!\}_p & \overset{\mathrm{def}}{=} & (\nu s)(\overline{p}s \mid \mathsf{OB}\langle \lambda(x_1,r).\{\![b_1]\!\}_r,\ldots,\lambda(x_n,r).\{\![b_n]\!\}_r, s\rangle)\\
\{\![a.l_j]\!\}_p & \overset{\mathrm{def}}{=} & (\nu q)(\{\![a]\!\}_q \mid q(x).\overline{x}l_j\_\mathsf{sel}\_p)\\
\{\![a.l_j \Leftarrow \varsigma(x_j)b]\!\}_p & \overset{\mathrm{def}}{=} & (\nu q)(\{\![a]\!\}_q \mid (\nu b)q(x).\\
 & & \quad \overline{x}l_j\_\mathsf{upd}\_\langle p, \lambda(x_j,r).\{\![b]\!\}_r\rangle)\\
\{\![x]\!\}_p & \overset{\mathrm{def}}{=} & \overline{p}x\\
\{\![\mathsf{clone}(a)]\!\}_p & \overset{\mathrm{def}}{=} & (\nu q)(\{\![a]\!\}_q \mid q(x).\overline{x}\mathsf{clone}\_p)\\
\{\![\mathsf{let}\ x = a\ \mathsf{in}\ b]\!\}_p^E & \overset{\mathrm{def}}{=} & (\nu q)(\{\![a]\!\}_q \mid q(x).\{\![b]\!\}_p)
\end{array}
$$

where OB is so defined:

$$
\begin{array}{l}
\mathsf{OB}(f_1 \ldots f_n, s) \ \overset{\mathrm{def}}{=}\\
\quad s\Big[_{j\in 1..n}\ l_j\_\big[\ \mathsf{sel}\_(x) \rhd f_j\langle s,x\rangle \mid \mathsf{OB}\langle f_1\ldots f_n, s\rangle;\\
\qquad\qquad\qquad \mathsf{upd}\_(x,y) \rhd \bar{x}s \mid \mathsf{OB}\langle f_1 \ldots f_{j-i}, y, f_{j+1}\ldots f_n, s\rangle \quad \big];\\
\qquad\qquad \mathsf{clone}\_(x) \rhd \mathsf{OB}\langle f_1\ldots f_n, s\rangle \mid (\nu s')(\bar{x}s' \mid \mathsf{OB}\langle f_1 \ldots f_n, s'\rangle)\Big]
\end{array}
$$

Table 3: The intermediate translation into HOπ (sketch)

The body of a method l_j is the function f_j; the set of these functions form the state of $\mathsf{OB}\langle f_1 \ldots f_n, s\rangle$.

We explain the behaviour of $\mathsf{OB}\langle f_1 \ldots f_n, s\rangle$ on operations of selection, update and cloning. In case of a select operation $l_j\_\mathsf{sel}\_p$ (which reads "activate method l_j and use p as location for the resulting object") the body f_j of method l_j is activated, with arguments $\langle s, p\rangle$; argument s, the access name of the value-core $\mathsf{OB}\langle f_1 \ldots f_n, s\rangle$, represents the self-parameter. An update request $l_j\_\mathsf{upd}\_\langle p, f\rangle$ (which reads "replace current method body for l_j with f, and use p as the location of the resulting object") results in a side effect on OB, whereby the j-th component of its state is updated to f. In a clone request clone_p (which reads "create a copy of the current object with location p"), a new object is created that has the same value-core $\mathsf{OB}\langle f_1 \ldots f_n, s\rangle$. Note the recursive definition of $\mathsf{OB}\langle f_1 \ldots f_n, s\rangle$, that shows that $\mathsf{OB}\langle f_1 \ldots f_n, s\rangle$ may accept arbitrarily many requests at s.

Now, following the translation of HOπ into π-calculus [23], we can turn the previous interpretation into a π-calculus one. For this, it suffices to make a recursive call with a functional argument, like $K\langle \lambda x.P\rangle$, into a first-order recursive call whose argument is a pointer to the function, like in:

$$(\nu b)(K\langle b\rangle | !b(x).P)$$

Correspondingly, a function application becomes an output of the arguments of the function along the pointer to the function. The result of this transformation, with the addition of type annotations, is presented in Table 4.

As for interpretation of Object Calculi into the λ-calculus [2], so our translation of terms has an environment E as parameter in order to put the necessary type annotations in the translation of method selection. This parameter could be avoided by having, for instance, more type information on the syntax of method selection. We assume that $p, q, r, b, s \ldots$ are not IOC variables.

In the remainder of the paper, we shall call processes of the form $\mathsf{OB}^A\langle b_1 \ldots b_n, s\rangle$ an *object manager* (in the interpretation of an object, $\mathsf{OB}^A\langle b_1 \ldots b_n, s\rangle$ acts like an administrator for the object; it "owns" the object methods, in the sense that it is the only process which can reach them, via names b_i's).

Finally we need to show how to translate types. The translation of an object type must be a type that specifies a repeated selection, update and clone operations.

$$\{\![\,[l_j{:}B_j\ ^{j\in 1..n}]\,]\!\} \overset{\mathrm{def}}{=} \mu X.\Big[_{j\in 1..n}\ l_j\_\big[\ \begin{array}{l}\mathsf{sel}\_\{\![B_j]\!\}^{\mathsf{w}^{\mathsf{w}}};\\ \mathsf{upd}\_\langle X^{\mathsf{w}^{\mathsf{w}}}, \langle X^{\mathsf{w}}, \{\![B_j]\!\}^{\mathsf{w}^{\mathsf{w}}}\rangle^{\mathsf{w}}\rangle\end{array}\ \big];\ \mathsf{clone}\_X^{\mathsf{w}^{\mathsf{w}}}\ \Big]$$

The pattern of occurrences of w tags is determined by the protocol which implements select and update operations. What is important, however, is the level of nesting of w tags: An even number of nesting gives covariance, whereas an odd number of nesting gives contravariance. Thus, the component $\{\![B_j]\!\}$ is in covariant position on selection, and in contravariance position on update: This explains the invariance of object types on the common components, in rule (OSub Obj) (the interpretation of IOC into the λ-calculus [2] does the same). Type environments are then interpreted componetwise:

$$\begin{array}{rcl}\{\![\emptyset]\!\} & \overset{\mathrm{def}}{=} & \emptyset\\ \{\![E, x{:}A]\!\} & \overset{\mathrm{def}}{=} & \{\![E]\!\}, x{:}\{\![A]\!\}^{\mathsf{w}}\end{array}$$

5 Simplifying the imperative part of the encoding

In the interpretation of IOC in Section 4, the key process is the object manager OB. This process is given as a recursive definition in which certain parameters may change during time. Having a state, this processes may be regarded as "imperative". In this section we modify the encoding, so that the only imperative processes are cell-like processes, each of which just stores a name. All remaining processes will be stateless, and therefore may be regarded as "functional". We thus obtain a clean separation of the interpretation into a functional part and a very simple imperative part. Technically, this factorisation allows us take full advantage of various π-calculus proof techniques for functional names and functional processes, discussed below.

$$
\begin{aligned}
\{\!\!\{[l_i{=}\varsigma(x_i{:}A)b_i\ {}^{i\in 1..n}]\}\!\!\}^E_p &\overset{\text{def}}{=} (\nu s{:}\{\!\!\{A\}\!\!\}^{\mathsf{b}})(\overline{p}s \mid (\nu b_i{:}T^{\mathsf{b}}_{A,i}\ {}^{i\in 1..n})(\mathsf{OB}^A\langle b_1 \dots b_n, s\rangle \mid \\
&\qquad \prod_{j\in 1..n} !b_j(x_j, r).\{\!\!\{b_j\}\!\!\}^{E,x_j:A}_r)) \\
\{\!\!\{a.l_j\}\!\!\}^E_p &\overset{\text{def}}{=} (\nu q{:}\{\!\!\{[l_j : B_j]\}\!\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!\!\{a\}\!\!\}^E_q \mid q(x).\overline{x}l_j\_\mathsf{sel}\_p) \\
\{\!\!\{a.l_j \Leftarrow \varsigma(x_j{:}A).b\}\!\!\}^E_p &\overset{\text{def}}{=} (\nu q{:}\{\!\!\{A\}\!\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!\!\{a\}\!\!\}^E_q \mid (\nu b{:}T^{\mathsf{b}}_{A,j})q(x). \\
&\qquad (\overline{x}l_j\_\mathsf{upd}\_\langle p, b\rangle \mid !b(x_j, r).\{\!\!\{b\}\!\!\}^{E,x_j:A}_r)) \\
\{\!\!\{x\}\!\!\}^E_p &\overset{\text{def}}{=} \overline{p}x \\
\{\!\!\{\mathsf{clone}(a)\}\!\!\}^E_p &\overset{\text{def}}{=} (\nu q{:}\{\!\!\{A\}\!\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!\!\{a\}\!\!\}^E_q \mid q(x).\overline{x}\mathsf{clone}\_p) \\
\{\!\!\{\mathsf{let}\ x{:}A = a\ \mathsf{in}\ b\}\!\!\}^E_p &\overset{\text{def}}{=} (\nu q{:}\{\!\!\{A\}\!\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!\!\{a\}\!\!\}^E_q \mid q(x).\{\!\!\{b\}\!\!\}^{E,x:A}_p)
\end{aligned}
$$

The object manager OB^A is so defined:

$$
\begin{aligned}
&\mathsf{OB}^A(b_1{:}T^{\mathsf{w}}_{A,1} \dots b_n{:}T^{\mathsf{w}}_{A,n}, s{:}\{\!\!\{A\}\!\!\}^{\mathsf{b}}) \overset{\text{def}}{=} \\
&\quad s\Big[_{j\in 1..n}\ l_j\_\big[\ \mathsf{sel}\_(x) \rhd \bar{b}_j\langle s, x\rangle \mid \mathsf{OB}^A\langle b_1 \dots b_n, s\rangle; \\
&\qquad\qquad \mathsf{upd}\_(x, y) \rhd \bar{x}s \mid \mathsf{OB}\langle b_1 \dots b_{j-i}, y, b_{j+1} \dots b_n, s\rangle\ \big]; \\
&\qquad \mathsf{clone}\_(x) \rhd \mathsf{OB}^A\langle b_1 \dots b_n, s\rangle \mid (\nu s' : \{\!\!\{A\}\!\!\}^{\mathsf{b}})(\bar{x}s' \mid \mathsf{OB}^A\langle b_1 \dots b_n, s'\rangle)\Big]
\end{aligned}
$$

and where

- $A = [l_j : B_j]$ and $T_{A,j} \overset{\text{def}}{=} \langle \{\!\!\{A\}\!\!\}^{\mathsf{w}}, \{\!\!\{B_j\}\!\!\}^{\mathsf{w}^{\mathsf{w}}}\rangle$
- in the encoding of selection, B_j is the unique type s.t. $E \vdash a{:}[\dots, l_j{:}B_j, \dots]$ holds, if one such judgement exists (the unicity of this type if a consequence of the minimum-type property of IOC), B_j can be any type otherwise;
- in the rule for update, x does not occour free in b.

Table 4: The interpretation of IOC into π-calculus

A π-calculus name is *functional* if its response to incoming messages does not change over time. Typically, a name a is functional if it only appears in subexpressions of the form $(\nu a)(!a(p).P|Q)$ where P and Q only possess the output capability on a, or of the form $(\nu a)(a(p).P|Q)$ if, in addition, a can only be used once in output. (In [25] names obeying these constraints are called *uniformly receptive.*)

Functional names have advantages. First, they can be implemented more efficiently than arbitrary names as done, for instance, in the language PICT [22]. Another advantage of functional names is their algebraic properties, among which copying or distributivity laws like

$$(\nu a)(!a(b).P|Q|R) = (\nu a)(!a(b).P|Q) \mid (\nu a)(!a(b).Q|R),$$

whose effect is to localise computation. Another property is τ-insensitiveness: Interactions along a functional name may not affect a process behaviour. As a consequence, when comparing the behaviour of two processes there are fewer configurations to take into account. Let us call a process *functional* if all inputs made by the process during its lifetime are at functional names. Functional languages (the λ-calculus, the Functional Object Calculus FOC) may be interpreted into a sublanguage of the π-calculus in which all processes are functional.

The new, factorised, encoding is defined in Table 5. To enhance readability we drop types, as they are the same as in the previous encoding. Only the clause for evaluated objects changes. Previously, the access name of the methods was part of (the state of) the object manager. By contrast, now a level of indirection is introduced, such that when updating a method, the indirection, instead of the object manager, changes; this way the object manager becomes functional.

A cell $\mathsf{Cell}\langle\iota, n\rangle$ stores a pointer n to a method; and can be accessed for read and write operations at ι. The cells are the only imperative processes in the encoding. Being imperative, a cell may be *shared* by several clients, but may not be *copied* among them. By contrast, all other resources are functional and they may be copied among their clients. The copy operator Copy is used to create new cells in a clone operation.

This factorised encoding is less compact, but has a simpler correctness proof, than the previous encoding of Section 4. In the next section, we use this factorised encoding for proving the correctness of the original one.

6 Correctness of the interpretation

6.1 Correctness of the interpretation of types

Theorem 2 (correctness for subtyping) *For all A, B, it holds that $\vdash A \leq B$ iff $\emptyset \vdash \{\!\!\{A\}\!\!\} \leq \{\!\!\{B\}\!\!\}$.*

Theorem 3 (correctness on type judgements)

1. *If $E \vdash a{:}A$ then, for all p, it holds that $\{\!\!\{E\}\!\!\}, p{:}\{\!\!\{A\}\!\!\}^{\mathsf{w}^{\mathsf{w}}} \vdash \{\!\!\{a\}\!\!\}_p^E$.*
2. *If $\{\!\!\{E\}\!\!\}, p{:}[\![A]\!]^{\mathsf{w}^{\mathsf{w}}} \vdash [\![a]\!]_p^E$, then $E \vdash a{:}A$.*

Below, we let $\tilde{\iota}$ denote $\iota_1 \ldots \iota_n$, and $\tilde{\iota}'$ denote $\iota'_1 \ldots \iota'_n$.

$$
\begin{aligned}
[\![[l_i{=}\varsigma(x_i{:}A)b_i \ ^{i \in 1..n}]]\!]_p \ \stackrel{\text{def}}{=}\ & (\nu s)(\bar{p}s \mid (\nu\tilde{\iota})(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \\
& \prod_{i \in 1..n} (\nu b_i)(\mathsf{Cell}\langle \iota_i, b_i\rangle \mid ! b_i(x_i, r).[\![b_i]\!]_r)))
\end{aligned}
$$

The other clauses are as for the original translation in Table 4.

The object manager is now defined thus:

$$
\begin{aligned}
& \mathsf{OB}_f(\tilde{\iota}, s) \ \stackrel{\text{def}}{=} \\
& \quad !s\Big[_{i \in 1..n} \ l_i\_\big[\ \mathsf{sel\_}(x) \triangleright (\nu g)(\overline{\iota_i}\mathsf{read\_}g.g(m).\overline{m}\langle s, x\rangle); \\
& \qquad\qquad\qquad \mathsf{upd\_}(x, y) \triangleright \overline{\iota_i}\mathsf{write\_}y.\overline{x}s) \quad \big]; \\
& \qquad\qquad\quad \mathsf{clone\_}(x) \triangleright (\nu\tilde{\iota}', s')(\mathsf{OB}_f\langle \tilde{\iota}', s'\rangle \mid \mathsf{Copy}^n\langle \tilde{\iota}, \tilde{\iota}', x, s'\rangle) \quad \Big]
\end{aligned}
$$

where $\mathsf{Cell}(\iota, m)$ and Copy^n $(n \geq 0)$ are so defined:

$$
\begin{aligned}
\mathsf{Cell}(\iota, m) \ & \stackrel{\text{def}}{=}\ \iota[\ \mathsf{read\_}(x) \triangleright \overline{x}m \mid \mathsf{Cell}\langle \iota, m\rangle; \\
& \qquad\ \mathsf{write\_}(y) \triangleright \mathsf{Cell}\langle \iota, y\rangle] \\
\mathsf{Copy}^n(\tilde{\iota}, \tilde{\iota}', x, s) \ & \stackrel{\text{def}}{=}\ (\nu g)\overline{\iota_n}\mathsf{read\_}g.g(m).(\mathsf{Cell}\langle \iota'_n, m\rangle \mid \mathsf{Copy}^{n-1}\langle \tilde{\iota}, \tilde{\iota}', x, s\rangle) \\
\mathsf{Copy}^0(\tilde{\iota}, \tilde{\iota}', x, s) \ & \stackrel{\text{def}}{=}\ \overline{x}s
\end{aligned}
$$

Table 5: The factorised encoding

6.2 Operational correctness

We first prove the operational correctness of the factorised encoding of Section 5; then we relate this encoding to the original one of Section 4.

Operational correctness of the factorised encoding. The proof of the correctness of the interpretation w.r.t. Abadi-Cardelli operational semantics has a few tricky points. First of all we need to extend the translation to deal with configurations (objects plus stores plus stacks) in the semantics. We therefore need to add translations of stores (σ), stacks ($\mathcal{S}$) and values (v). We found it quite difficult to make such an extension of the original encoding $\{\![-]\!\}$ of Section 4, due to sharing of closures. We solve this problem by using instead the factorised encoding $[\![-]\!]$ of Section 5. A location ι of a store holds a method closure (which is a method together its stack); we translate it using a cell located at ι that holds the address of the method closure (which is the translation of the method together with a private stack).

$$\begin{aligned}
[\![\emptyset]\!] &\stackrel{\text{def}}{=} \mathbf{0}\\
[\![\iota \mapsto \langle \varsigma(x)b, \mathcal{S}\rangle : \sigma]\!] &\stackrel{\text{def}}{=} (\nu \mathsf{dom}(\mathcal{S}), m)(\mathsf{Cell}\langle \iota, m\rangle \mid !m(x,r).[\![b]\!]_r \mid [\![\mathcal{S}]\!]) \mid [\![\sigma]\!]
\end{aligned}$$

A stack binds values to variables, so we translate it as an object manager located on the variable name.

$$\begin{aligned}
[\![\emptyset]\!] &\stackrel{\text{def}}{=} \mathbf{0}\\
[\![x \mapsto [l_i{=}\iota_i\ {}^{i\in 1..n}] : \mathcal{S}]\!] &\stackrel{\text{def}}{=} \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, x\rangle \mid \mathcal{S}
\end{aligned}$$

We have two types of configurations in the operational semantics for Imperative Object Calculus; initial configurations on the form $\sigma \cdot \mathcal{S} \vdash a$ and final configurations on the form $v \cdot \sigma$. They are translated as follows:

$$\begin{aligned}
[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p &\stackrel{\text{def}}{=} (\nu \mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu \mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid [\![a]\!]_p))\\
[\![[l_i{=}\iota_i\ {}^{i\in 1..n}] \cdot \sigma]\!]_p &\stackrel{\text{def}}{=} (\nu s, \mathsf{dom}(\sigma))(\bar{p}s \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid [\![\sigma]\!])
\end{aligned}$$

That is, in both cases we simply translate each component of the configurations and appropriately hide their access. Below, $\sim$ is the strong version of barbed congruence.

Theorem 4 *If* $\sigma \cdot \mathcal{S} \vdash a \leadsto v \cdot \sigma'$ *then:* $[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p \Longrightarrow_d \sim\ [\![v \cdot \sigma']\!]_p$.

Lemma 5 *Suppose* a *is well-typed. If* $a\Uparrow$ *then* $[\![\emptyset \cdot \emptyset \vdash a]\!]_p$ *has an infinite computation* $[\![\emptyset \cdot \emptyset \vdash a]\!]_p \xrightarrow{\tau}_d P_1 \ldots \xrightarrow{\tau}_d P_n \xrightarrow{\tau}_d \ldots$.

Theorem 6 *Suppose* a *is well-typed. If* $[\![\emptyset \cdot \emptyset \vdash a]\!]_p \Downarrow_p$, *there exists a value* v, *and a store* σ, *s.t.* $\sigma \cdot \mathcal{S} \vdash a \leadsto v \cdot \sigma$

Relating the original and the factorised encodings. We have not been able to prove that the translations of an IOC object, according to the factorised encoding $\{\!|-|\!\}$ and to the original encoding $[\![-]\!]$ are equated by some known behavioural equivalence of the π-calculus — the factorised encoding yields richer behaviours. However, we have been able to extend the notion of *ready simulation* [4, 13] to the π-calculus and prove that the two translations of an object are in a ready simulation relation. This is a rather weak result, but, together with the correctness of the factorised encoding, it will suffice to prove the operational correctness of the original encoding.

Definition 7 (Ready simulation) Ready simulation *is the largest relation* $\prec$ *s.t.* $P \prec Q$ *implies:*

1. *If* $P \xrightarrow{\mu} P'$; *then there exist a* Q' *s.t.* $Q \stackrel{\hat{\mu}}{\Longrightarrow} Q'$ *and* $P' \prec Q'$. *With* $\stackrel{\hat{\mu}}{\Longrightarrow} \stackrel{def}{=} \Longrightarrow$ *if* $\mu = \tau$ *and* $\stackrel{\mu}{\Longrightarrow}$ *otherwise.*

2. *If* $Q \xrightarrow{\mu}$*; then* $P \overset{\mu}{\Longrightarrow}$.

Theorem 8 $\{\!\!\{a\}\!\!\}_p \prec \llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p$.

Lemma 9 *Suppose* a *is well-typed. Then* $\{\!\!\{a\}\!\!\}_p$ *cannot deadlock, i.e. whenever* $\{\!\!\{a\}\!\!\}_p \Longrightarrow P$ *then there are* μ, P' *s.t.* $P \xrightarrow{\mu} P'$.

Proof. Assume that $\{\!\!\{a\}\!\!\}_p$ can deadlock, that is there exists a P such that $\{\!\!\{a\}\!\!\}_p \Longrightarrow P \overset{\mu}{\not\rightarrow}$. Now since $\{\!\!\{a\}\!\!\} \prec \llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p$ there must exist a Q such that $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p \Longrightarrow Q$ with $P \prec Q$.

By Theorem 4 if a converges then $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p$ will also converge and by Lemma 5 if a diverges then $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p$ will also diverge; so $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p$ cannot deadlock. Then there must exist a μ s.t. $Q \xrightarrow{\mu}$. By the second clause of the definition of ready simulation we infer $P \overset{\mu}{\Longrightarrow}$, which is a contradiction. $\square$

Corollary 10 *Suppose* a *is well-typed.*

- $\{\!\!\{a\}\!\!\}_p\Uparrow$ *iff* $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p\Uparrow$
- $\{\!\!\{a\}\!\!\}_p\Downarrow$ *iff* $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p\Downarrow$

Proof. Both $\{\!\!\{a\}\!\!\}_p$ and $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p$ are deadlock-free. Then the corollary follows from Theorem 8 and the fact that $\llbracket\emptyset \cdot \emptyset \vdash a\rrbracket_p$ either converges or diverges (it cannot do both, because of Theorem 4 and Lemma 5). $\square$

Adequacy and soundness of the original interpretation. From Theorems 4 and 6 and Corollary 10, we infer:

Corollary 11 (computational adequacy) *If* $\emptyset \vdash a : A$ *for some type* A*, then* $a\Downarrow$ *iff* $\{\!\!\{a\}\!\!\}_p\Downarrow_p$.

Behavioural equivalences like barbed congruence or the Morris-style contextual equivalence can also be defined in IOC (in fact, on IOC the two equivalences coincide). We write $a \simeq_B b$ if a and b are closed terms of IOC of type B and are barbed congruent.

We can show soundness of the translation using compositionality of the encoding and adequacy. This tells us that the equalities that can be proven using the translation are valid equalities.

Theorem 12 (soundness) *Assume* $a{:}A$ *and* $b{:}A$*. If* $\{\!\!\{a\}\!\!\}_p \approx_{p:\{\!\!\{A\}\!\!\}^{\mathrm{ww}}} \{\!\!\{b\}\!\!\}_p$ *then* $a \simeq_A b$.

As for the encodings of the λ-calculi into π-calculus, so in the case of IOC the converse of soundness does not hold.

7 Comparisons with the interpretation of FOC

In their book, Abadi and Cardelli consider not only the imperative, but also the functional paradigm for Object Calculi. In this section we briefly compare the (first-order) Imperative and Functional Object Calculi, and their encodings into the π-calculus. The syntax of the *Functional Object Calculus* (FOC) is the same as for the imperative except that we do not have the the let and clone constructs. The operational semantics, however, is very different. In the functional case, stores and stacks are not necessary and a simple reduction semantics can be given, using the rules below, where $a = [l_i{=}\varsigma(x_i{:}A_i)b_i \ ^{i\in 1..n}]$:

$$\begin{array}{lll} a.l_k & \leadsto \ b_k\{^{a}\!/\!_{x_k}\} & (k \in 1..n) \\ a.l_k{\Leftarrow}\varsigma(x{:}A)b & \leadsto \ [l_k{=}\varsigma(x{:}A)b,\ l_i{=}\varsigma(x_i{:}A_i)b_i \ ^{i\in 1..n\setminus\{k\}}] & (k \in 1..n) \end{array}$$

As a consequence of the differences in the operational semantics, certain basic laws of FOC, like $a.l_k \ = \ b_k\{^{a}\!/\!_{x_k}\}$ $(k \in 1..n)$, do not hold in IOC.

In [24] FOC is translated onto the same typed π-calculus we use in this paper. Remarkably, despite the strong differences in the operational semantics, the interpretations of IOC and FOC into the π-calculus are structurally very close. Roughly, the only difference between the two translations is in the object manager. The FOC manager, reported below, is a functional process (it is replicated, see the discussion on functional processes in Section 5). This difference has consequences on the update requests: On such a request, the FOC manager always generates a new object manager, whereas the IOC manager works by having side effect on itself.

$$\begin{array}{l} \mathsf{OB}^{A}(b_1{:}T^{\mathsf{w}}_{A,1} \ldots b_n{:}T^{\mathsf{w}}_{A,n}, s{:}\{\!\!\{A\}\!\!\}^{\mathsf{b}}) \ \stackrel{\mathrm{def}}{=} \\ !s\Big[_{j\in 1..n} \ l_j\_[\ \mathsf{sel\_}(x) \rhd \bar{b}_j\langle s, x\rangle; \\ \qquad\qquad\qquad \mathsf{upd\_}(x,y) \rhd (\nu s'{:}\{\!\!\{A\}\!\!\}^{\mathsf{b}})(\overline{x}s' \mid \mathsf{OB}\langle b_1 \ldots b_{j-i}, y, b_{j+1} \ldots b_n, s'\rangle) \quad]\Big] \end{array}$$

The functional and the imperative nature of, respectively, FOC and IOC, is reflected in the functional and the imperative nature of the object managers of the π-calculus interpretations. The commonalities between the interpretations of FOC and IOC allow us to reuse certain proofs, most notably those on types (see Theorem 2 and 3), but also certain proofs about behavioural properties of objects (a good example of this is the proof of the law (EQ SUB OBJ), in Section 8).

8 Reasoning about objects

In this section we give some examples of how the π-calculus interpretation can be used to validate some basic behavioural properties of IOC.

Lemma 13 *Let* $o = [l_i{=}\varsigma(x_i{:}A)b_i \ ^{i\in 1..n}]$, $A = [l_i{:}B_i \ ^{i\in 1..n}]$, *and* $o{:}A$. *Then*

1. $o.l_j \simeq_{B_i} \mathsf{let}\ x_j{:}A = o\ \mathsf{in}\ b_j$
2. *If* $x{:}A \vdash b{:}B_j$ *then* $o.l_j \Leftarrow\varsigma(x{:}A)b \simeq_A [l_j{=}\varsigma(x{:}A)b,\ l_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n\setminus\{j\}}]$
3. $\mathsf{clone}(o) \simeq_A o$
4. *If* $a\Downarrow$, $a{:}B$, $b{:}C$ $x \notin \mathsf{fv}(b)$, *then* $\mathsf{let}\ x{:}B = a\ \mathsf{in}\ b \simeq_C b$.
5. *If* $x \notin \mathsf{fv}(o)$, $y \notin \mathsf{fv}(a)$, $a{:}B$ *and* $(\mathsf{let}\ x{:}B = a\ \mathsf{in}\ \mathsf{let}\ y{:}A = o\ \mathsf{in}\ b) : C$, *then*

   $$(\mathsf{let}\ x{:}B = a\ \mathsf{in}\ \mathsf{let}\ y{:}A = o\ \mathsf{in}\ b) \simeq_C (\mathsf{let}\ y{:}A = o\ \mathsf{in}\ \mathsf{let}\ x{:}B = a\ \mathsf{in}\ b)$$

6. *(law (*Eq Sub Object*)): If* $m \geq n$, *then* $o \simeq_A [l_i{=}\varsigma(x_i{:}B)b_i\ ^{i\in 1..m}]$.

Laws 1-5 can be validated using the theory of the untyped π-calculus. For instance, using laws such as the expansion law, $(\nu p)(p(x).P \mid \overline{p}v.Q) \approx (\nu p)(P\{v/x\}|Q)$ and $(\nu p)(!p(x).P|Q) \approx Q$ if p is not free in Q, we can prove law (5) as follows.

$$\begin{array}{rl}
& \{\!\{\mathsf{let}\ y = o\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ b\}\!\}_p \\
= & (\nu q)((\nu s, b_1 \dots b_n)(\overline{q}s \mid \mathsf{OB}\langle b_1 \dots b_n, s\rangle \mid \prod_{i\in 1..n} !b_i(x_i,r).\{\!\{b_i\}\!\}_r) \mid \\
& \quad q(y).(\nu q')([\![a]\!]_{q'} \mid q'(x).\{\!\{b\}\!\}_p)) \\
\approx & (\nu q')(\{\!\{a\}\!\}_{q'} \mid q'(x).(\nu s, b_1 \dots b_n)(\{\!\{b\}\!\}\{s/y\} \mid \mathsf{OB}\langle b_1 \dots b_n, s\rangle \mid \\
& \quad \prod_{i\in 1..n} !b_i(x_i,r).\{\!\{b_i\}\!\}_r)) \\
\approx & (\nu q')([\![a]\!]_{q'} \mid q(x).(\nu q)((\nu s)(\overline{q}s \mid \mathsf{OB}\langle b_1 \dots b_n, s\rangle \mid \\
& \quad \prod_{i\in 1..n} !b_i(x_i,r).\{\!\{b_i\}\!\}_r) \mid q(y).\{\!\{b\}\!\}_p)) \\
= & \{\!\{\mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = o\ \mathsf{in}\ b\}\!\}_p
\end{array}$$

It is interesting to look at the difference between FOC and IOC on objects of the form $o.l_j$, for $o = [l_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}]$ $(j \in 1..n)$, using the π-calculus interpretations of FOC and IOC . In the case of FOC, o is interpreted as a functional process and we can therefore apply copy laws to derive $\{\!\{o.l_j\}\!\}_p \approx \{\!\{b_j\{a/x_j\}\}\!\}_p$. By contrast, in the case of IOC, object o is interpreted as a process with a state, and we can only infer $\{\!\{o.l_j\}\!\}_p \approx \{\!\{\mathsf{let}\ x_j{:}A = o\ \mathsf{in}\ b_j\}\!\}_p$, as by Lemma 13(1).

Law (Eq Sub Object) of Lemma 13 can be validated using the typed labeled bisimulation technique. Roughly, a typed bisimulation relations consists of triples (Γ, P, Q), and (Γ, P, Q) being in a typed bisimulation means that P and Q are undistinguishable by an observer whose use of names respect the type informations in Γ. We describe a typed bisimulation for proving (Eq Sub Object). Having fixed a and b, their translations are:

$$\begin{array}{rcl}
[\![a]\!]_p & = & (\nu s{:}[\![A]\!]^{\mathsf{w}})(\overline{p}s \mid (\nu b_i{:}T^{\mathsf{b}}_{A,i}\ ^{i\in 1..n})(\mathsf{OB}^{A}\langle s, b_1 \dots b_n\rangle \mid \prod_{i\in 1..n} !b_i(x_i,r).[\![b_i]\!]_r)) \\
[\![b]\!]_p & = & (\nu s{:}[\![A']\!]^{\mathsf{w}})(\overline{p}s \mid (\nu b_i{:}T^{\mathsf{b}}_{A,i}\ ^{i\in 1..m})(\mathsf{OB}^{A'}\langle s, b_1 \dots b_m\rangle \mid \prod_{i\in 1..m} !b_i(x_i,r).[\![b_i]\!]_r))
\end{array}$$

Now, let $\Gamma' = s{:}\llbracket A\rrbracket^{\mathsf{w}}, b_1{:}T^{\mathsf{r}}_{A,1}, \ldots, b_n{:}T^{\mathsf{r}}_{A,n}$ and $\Gamma'' = p{:}\llbracket A\rrbracket^{\mathsf{w}^{\mathsf{r}}}, b_1{:}T^{\mathsf{r}}_{A,1}, \ldots, b_n{:}T^{\mathsf{r}}_{A,n}$. where $T_{A,j} \overset{\mathrm{def}}{=} \langle \llbrace A\rrbrace^{\mathsf{w}}, \llbrace B_j\rrbrace^{\mathsf{w}^{\mathsf{w}}}\rangle$. Consider the relation consisting of these two triples:

$$1)\qquad \Big(\Gamma', \mathsf{OB}^{A}\langle b_1 \ldots b_n, s\rangle,$$
$$(\nu b_i{:}T^{\mathsf{b}}_{A,i}\ {}^{i\in n+1..m})(\mathsf{OB}^{A'}\langle b_1 \ldots b_m, s\rangle \mid \prod_{i\in n+1..m} !b_i(x_i, r).\llbracket b_i\rrbracket_r)\Big)$$

$$2)\qquad \Big(\Gamma'', (\nu s{:}\llbracket A\rrbracket^{\mathsf{w}})(\bar{p}s \mid \mathsf{OB}^{A}\langle b_1 \ldots b_n, s\rangle,$$
$$(\nu s{:}\llbracket A'\rrbracket^{\mathsf{w}})(\nu b_i{:}T^{\mathsf{b}}_{A,i}\ {}^{i\in n+1..m})(\bar{p}s \mid \mathsf{OB}^{A'}\langle b_1 \ldots b_m\rangle \mid \prod_{i\in n+1..m} !b_i(x_i, r).\llbracket b_i\rrbracket_r)\Big)$$

This relation is typed bisimulation up to parallel composition and up to injective substitutions.

This is the first proof of (Eq Sub Object) for IOC we are aware of. The proof can be easily adapted to the analogous law for FOC.

9 Extensions and further work

Our interpretation of IOC can be extended to accommodate other type features discussed in [1], like variant tags, recursive types, polymorphic types. *Variant tags* are tags on method names which allow only selection or update operations on a method, so to have a richer subtyping relation. These tags yield the same form of subtyping on IOC types as that induced by the tags $\{\mathsf{r}, \mathsf{w}, \mathsf{b}\}$ on the π-calculus types. We can capture them with a simple refinement of the encoding of types. For *recursive types*, we just map type variables of a IOC type to type variables of the π-calculus types. Similarly, is possible to handle *polymorphic types*, using polymorphic extensions of the π-calculus [26].

The extensibility of the interpretation of imperative objects, and its resemblance to the interpretation of functional objects, suggests that the representation of objects into the π-calculus is a robust one, and that it could be used for giving the semantics to, and proving properties of, a wide range of object-oriented languages, possibly combining imperative, functional and concurrent features. We have began examining the case of Obliq [6], which has imperative objects and constructs for concurrency and distribution.

We hope the study of the Object Calculi from within the π-calculus will help to develop concurrent or distributed versions of the Object Calculi.

An interesting and challenging question is finding a direct characterisation of the equivalence induced on IOC terms by the encoding into the π-calculus (where two IOC terms are equated if their process translations are behaviourally equivalent).

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer Verlag, 1996.

[2] M. Abadi, L. Cardelli, and R. Viswanathan. An interpretation of Objects and Objects Types. In *Proc. 23th POPL.* ACM Press, 1996.

[3] L. Aceto, H. Hüttel, A. Ingólfsdóttir, and J. Kleist. Relating semantic models for the object calculus. In *Proceedings of Express 97 Workshop*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 1997.

[4] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of the Association for Computing Machinery*, 42(1):232–268, Jan. 1995.

[5] M. Boreale and D. Sangiorgi. Bisimulation without matching. To appear.

[6] L. Cardelli. Obliq — a language with distributed scope. Research report 122, Digital Equipment Coporation, Systems Research Center, 1994.

[7] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proceedings of 17th FST&TCS Conference*, 1997.

[8] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, 1996.

[9] K. Honda. Composing processes. In *Proc. 23th POPL.* ACM Press, 1996.

[10] H. Hüttel and J. Kleist. Objects as mobile processes. Technical report RR-96-38, BRICS - Basic Research in Computer Science, 1996.

[11] C. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 1993.

[12] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. In *Proc. 23th POPL.* ACM Press, 1996.

[13] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, Sept. 1991.

[14] X. Liu and D. Walker. Partial confluence of processes and systems of objects. Submitted for publication, 1996.

[15] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[16] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[17] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., Oct. 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[19] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[20] A. Philippou and D. Walker. On transformations of concurrent object programs. In *Proc. CONCUR '96*, Lecture Notes in Computer Science, pages 131–147. Springer Verlag, 1996.

[21] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of LICS'93*, pages 376–385. IEEE Computer Society Press, 1993.

[22] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Indiana University Technical report, 1997., 1997.

[23] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[24] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report RR-3000, INRIA — Sophia Antipolis, 1996.

[25] D. Sangiorgi. The name discipline of uniform receptiveness. In *Proceedings of ICALP'97*, 1997.

[26] D.N. Turner. *The polymorphic pi-calculus: Theory and Implementation.* PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[27] F. Vaandrager. A process algebra semantics of POOL. In *Applications of process algebra*, volume 17 of *Tracts in Theoretical Computer Science*, pages 173–236. Cambridge University Press, 1990.

[28] V. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Proc. Object Technologies for Advanced Software '93*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer Verlag, 1993.

[29] D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

A Grammar and rules for IOC

Syntax:

Type Environments	E	$::= \emptyset \mid E, x : A$	
Types	A, B	$::= [l_i{:}B_i \ ^{i\in 1..n}]$	
Method names	$l,$		
Variables	x, y, z		
Terms	a, b	$::=$	
		x	Variable
	$\mid$	$[l_i{=}\varsigma(x_i{:}A)b_i \ ^{i\in 1..n}]$	Object
	$\mid$	$a.l$	Method activation
	$\mid$	$a.l{\Leftarrow}\varsigma(x{:}A)b$	Method override
	$\mid$	$\mathsf{clone}(a)$	Cloning
	$\mid$	$\mathsf{let}\ x{:}A = a\ \mathsf{in}\ b$	Local definition

Operational semantics:

(OC Var)

$$\frac{\sigma \cdot \mathcal{S} \vdash \diamond \quad x \in \mathsf{dom}(\mathcal{S})}{\sigma \cdot \mathcal{S} \vdash x \leadsto \mathcal{S}(x) \cdot \sigma}$$

(OC Obj) where $\sigma' = (\sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, \mathcal{S}\rangle \ ^{i\in 1..n})$

$$\frac{\sigma \cdot \mathcal{S} \vdash \diamond \quad \iota_i \notin \mathsf{dom}(\sigma) \quad \iota_i \text{ distinct } \forall i \in 1..n}{\sigma \cdot \mathcal{S} \vdash [l_i{=}\varsigma(x_i{:}A)b_i \ ^{i\in 1..n}] \leadsto [l_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma'}$$

(OC Sel)

$$\frac{\begin{array}{c}\sigma \cdot \mathcal{S} \vdash a \leadsto [l_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma' \quad \sigma'(\iota_j) = \langle \varsigma(x_j)b_j, \mathcal{S}'\rangle \\ x_j \notin \mathsf{dom}(\mathcal{S}') \quad j \in 1..n \quad \sigma' \cdot \mathcal{S}', x_j \mapsto [l_i{=}\iota_i \ ^{i\in 1..n}] \vdash b_j \leadsto v \cdot \sigma''\end{array}}{\sigma \cdot \mathcal{S} \vdash a.l_j \leadsto v \cdot \sigma''}$$

(OC Upd)

$$\frac{\sigma \cdot \mathcal{S} \vdash a \leadsto [l_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma' \quad j \in 1..n \quad \iota_j \in \mathsf{dom}(\sigma')}{\sigma \cdot \mathcal{S} \vdash a.l_j{\Leftarrow}\varsigma(x{:}A)b \leadsto [l_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma'[\iota_j \mapsto \langle \varsigma(x)b, \mathcal{S}\rangle]}$$

(OC Clone)

$$\frac{\sigma \cdot \mathcal{S} \vdash a \leadsto [l_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma' \quad \forall i \in 1..n \quad \iota'_i \notin \mathsf{dom}(\sigma') \quad \iota'_i \text{ distinct}}{\sigma \cdot \mathcal{S} \vdash \mathsf{clone}(a) \leadsto [l_i{=}\iota_i \ ^{i\in 1..n}] \cdot (\sigma', \iota'_i \mapsto \sigma(\iota_i) \ ^{i\in 1..n})}$$

(OC Let)

$$\frac{\sigma \cdot \mathcal{S} \vdash a \leadsto v' \cdot \sigma' \quad \sigma' \cdot \mathcal{S}, x \mapsto v' \vdash b \leadsto v'' \cdot \sigma''}{\sigma \cdot \mathcal{S} \vdash \mathsf{let}\ x{:}A = a\ \mathsf{in}\ b \leadsto v'' \cdot \sigma''}$$

Subtyping rules:

(OSub Obj)

$$\frac{}{[l_i{:}B_i \ ^{i\in 1..n+m}] <: [l_i{:}B_i \ ^{i\in 1..n}]}$$

Typing rules: (the order of assignments in a type environment is ignored)

(OT SUBSUMPTION)

$$\frac{E \vdash a{:}A \quad A <: B}{E \vdash a{:}B}$$

(OT VAR) where $x \in \mathsf{dom}(E)$

$$\frac{E(x) = A}{E \vdash x{:}A}$$

(OT OBJ) where $A = [l_i{:}B_i \ ^{i \in 1..n}]$

$$\frac{E, x_j{:}A \vdash b_j{:}B_j \quad \forall j \in 1..n}{E \vdash [l_i{=}\varsigma(x_i{:}A)b_i \ ^{i \in 1..n}] : A}$$

(OT SEL) where $A = [l_i{:}B_i \ ^{i \in 1..n}]$

$$\frac{E \vdash a : [l_i{:}B_i \ ^{i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j{:}B_j}$$

(OT UPD) where $A = [l_i{:}B_i \ ^{i \in 1..n}]$

$$\frac{E \vdash a{:}A \quad E, x{:}A \vdash b{:}B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A}$$

(OT CLONE)

$$\frac{E \vdash a{:}A}{E \vdash \mathsf{clone}(a){:}A}$$

(OT LET)

$$\frac{E \vdash a{:}A \quad E, x{:}A \vdash b{:}B}{E \vdash \mathsf{let}\ x{:}A = a\ \mathsf{in}\ b : B}$$