An Interpretation of Typed Concurrent Objects in the Blue Calculus – DRAFT –

Silvano Dal-Zilio[⋆]

Microsoft Research

Abstract. We propose an interpretation of a typed concurrent calculus of objects (**conc**$\boldsymbol{\varsigma}$) based on the model of Abadi and Cardelli's imperative object calculus. The target of our interpretation is a version of the blue calculus, a variant of the π-calculus that directly contains the λ-calculus, with record and first-order types. We show that reduction and type judgements can be derived in a rather simple and natural way, and that our encoding can be extended to "self-types" and synchronisation primitives. We also prove some equational laws on objects.

1 Introduction

In the recent past, there has been a growing interest in the theoretical foundations of object-oriented and concurrent programming languages. One of the means used to explain object-oriented concepts, such as object types or self-referencing for example, has been to look for an interpretation of these concepts in simpler formalisms, such as typed λ-calculi. But these interpretations are difficult, and very technical, due to the difficulties raised by the typing (and subtyping) of objects. To circumvent these problems, Abadi and Cardelli have defined a "canonical" object-oriented calculus, the $\boldsymbol{\varsigma}$-calculus [1], in which the notion of object is primitive, and they have developed and studied type systems for this calculus.

The purpose of this paper is to give a model of concurrent object computation based on a modelling of "objects as processes". To this end, we introduce some derived notations for objects and we give their translation in a version of the blue calculus ($\boldsymbol{\pi}^\star$) [8] extended with records. Types for blue processes are given in an implicit type system based on the simply typed λ-calculus. Then, we give an interpretation of a concurrent and imperative version of $\boldsymbol{\varsigma}$, denoted **conc**$\boldsymbol{\varsigma}$, which has been defined by Gordon and Hankin [20]. This interpretation preserves reduction, typing and subtyping judgements. Moreover, the encoding gives an interpretation of complex notions, such as method update or object types, in terms of more basic notions such as records, field selection and functional types. A consequence is that we obtain a type-safe way to implement higher-order concurrent objects in the blue calculus, and therefore in $\boldsymbol{\pi}$.

The structure of the paper is as follows. In the next section, we introduce the blue calculus using very simple intuitions taken from the λ-calculus execution model. This is an occasion to give an informal and intuitive presentation of the blue calculus to the reader. In Sect. 3 and 4, we briefly introduce Gordon and Hankin's calculus of objects and we define its interpretation in $\boldsymbol{\pi}^\star$. We prove that **conc**$\boldsymbol{\varsigma}$ is embedded in $\boldsymbol{\pi}^\star$ and that objects can be viewed as a particular kind of (linearly managed) resources. Sect. 5.2 is dedicated to the typing of processes and objects. It introduces a new type operator that is very well suited for typed continuation

[⋆] Email: sdal@microsoft.com. Address: Microsoft Research Ltd, St George House, 1 Guildhall St., Cambridge, CB2 3NH – England. Fax: (+44) 1223 744 777.

passing style transformations. Before concluding, we present two possible applications of our interpretation.

2 The Blue Calculus

In the functional programming world, a program is ideally represented by a λ-calculus term, that is a term generated by the following grammar:

$$M, N ::= x \mid \lambda x.M \mid (MN)$$

We enrich this calculus with a set of constants: $a_1, \dots a_n$, that we call names, and that can be interpreted as resource locations. We describe a very simple execution model for programs written in this syntax, based on the notion of abstract machine (AM), and we enrich it until we obtain a model that exhibits concurrent behaviours similar to those expressible in the π-calculus ($\boldsymbol{\pi}$). Going back from this AM, we obtain the blue calculus. Therefore $\boldsymbol{\pi}^\star$ can intuitively be viewed, at the same time, as a concurrent λ-calculus and as an applicative π-calculus.

2.1 Concurrent Chemical Machine

An abstract machine is defined by a set of *configurations*, denoted $\mathcal{K}$, and a set of *transition rules*, $\mathcal{K} \to \mathcal{K}'$, which define elementary computing steps. In our setting, a machine configuration is a triple $\{\mathcal{E}; M; \mathcal{S}\}$ where:

- $\mathcal{E}$ is a memory, also called *environment*, that is an association between names and programs;
- M is a program, that is a λ-term;
- $\mathcal{S}$ is a stack containing the arguments of functional calls.

Initially an AM is supposed to have an empty memory, denoted by the symbol ϵ, which can be extended with new *declarations*. This operation is denoted $(\mathcal{E} \mid \langle a_n {=} N \rangle)$. The stack has a similar structure, except that it keeps names and not declarations.

$$\mathcal{E} ::= \epsilon \mid (\mathcal{E} \mid \langle a_n {=} N \rangle) \qquad\qquad \mathcal{S} ::= \epsilon \mid (a_j, \mathcal{S})$$

An execution of the functional AM start in the initial configuration $\mathcal{K}_0$, with an empty stack and memory, $\mathcal{K}_0 = \{\epsilon; M; \epsilon\}$. We describe an execution as a sequence of elementary computing step, or transitions, denoted $\mathcal{K} \to \mathcal{K}'$.

Definition 2.1 (Functional Machine). *The functional AM transitions are defined by the following set of rules, where $M\{^{a_j}/_{x}\}$ is the substitution of the variable x by the name a_j in M:*

$$\begin{array}{rcll} \{\mathcal{E}; a_j; \mathcal{S}\} & \to & \{\mathcal{E}; N_j; \mathcal{S}\} & (\mathcal{E} = \cdots \mid \langle a_j {\Leftarrow} N_j \rangle \mid \cdots) \\ \{\mathcal{E}; \lambda x.M; (a_j, \mathcal{S})\} & \to & \{\mathcal{E}; M\{^{a_j}/_{x}\}; \mathcal{S}\} & \\ \{\mathcal{E}; (MN); \mathcal{S}\} & \to & \{(\mathcal{E} \mid \langle a_n {=} N \rangle); M; (a_n, \mathcal{S})\} & (a_n \textit{ fresh name}) \end{array}$$

Therefore, to evaluate a function application for example, we memorise the argument in a fresh memory location, and we add the name of this location in the stack. Note that the reduction is call-by-name [34].

At each computation step, the machine is in a configuration of the kind:

$$\mathcal{K}_n = \{(\langle \mathrm{a}_1{=}N_1\rangle \mid \cdots \mid \langle a_n{=}N_n\rangle); M; (a_{i_1}, \dots, a_{i_k})\}$$

where the indices $i_1, \dots, i_k$ are in the interval $[1..n]$. Each configuration corresponds to a λ-term and, for example, the configuration $\mathcal{K}_n$ corresponds to $(M a_{i_1} \dots a_{i_k})\{{}^{N_1}\!/_{a_1}\} \dots \{{}^{N_n}\!/_{a_n}\}$. Therefore, to each extension of the functional AM, there corresponds a generalisation of the λ-calculus. In the remainder of this section, we improve the functional AM until we obtain an execution model that compares to the one of $\boldsymbol{\pi}$. The calculus defined by the extended AM is the blue calculus [8].

We start with simple syntactical modifications. More precisely, we modify the notation and use a sequence of applications instead of a stack, and we rewrite the standard configuration into: $\langle a_1{=}N_1\rangle \mid \cdots \mid \langle a_n{=}N_n\rangle \mid (M a_{i_1} \dots a_{i_k})$. With these modifications, the transition rules can be reformulated in the following way:

$$\begin{array}{rcll}
\langle a{=}N\rangle \mid \cdots \mid (a a_1 \dots a_n) & \to & \langle a{=}N\rangle \mid \cdots \mid (N a_1 \dots a_n) & (\rho)\\
(\lambda x.M) a_1 \dots a_n & \to & M\{{}^{a_1}\!/_{x}\} a_2 \dots a_n & (\beta)\\
(MN) a_1 \dots a_n & \to & \langle a{=}N\rangle \mid M a a_1 \dots a_n & (\chi)\\
\mathcal{K} \to \mathcal{K}' & \Rightarrow & (\langle a{=}N\rangle \mid \mathcal{K}) \to (\langle a{=}N\rangle \mid \mathcal{K}') & (\varpi)
\end{array}$$

In order to obtain a model "equivalent" to the one defined by the functional AM, we suppose that the name a is fresh in rule (χ).

In this new presentation, rule (β) corresponds to a simplified form of beta-reduction, where we substitute a name, and not a term, to a variable, whereas rule (ρ) can be interpreted as a form of communication. Nonetheless, whereas the classical π-calculus communication model is based on message synchronisation, we use instead a particular kind of resource fetching.

Adding Parallelism and Non-Determinism. A first improvement to the AM is to consider $\mid$ as an associative composition operator, and to allow multiple configurations in parallel. We do not choose a commutative operator though. The idea is to separate, in each configuration, the store from the active part, that is to separate the memory from the term being evaluated. Nonetheless we allow some commutations, with the restriction that the evaluated term M is always at the right of the topmost parallel composition. More formally, we consider the following "structural rules" for the operator $\mid$, where $P \leftrightarrows Q$ means that both $P \to Q$ and $Q \to P$ holds.

$$(M_1 \mid M_2) \mid M_3 \leftrightarrows M_1 \mid (M_2 \mid M_3) \quad \text{and} \quad (M_1 \mid M_2) \mid M_3 \leftrightarrows (M_2 \mid M_1) \mid M_3$$

As a result, we obtain an asymmetric parallel composition operator, like the one defined in **concς**, or the formal description of CML [17]. Another consequence is that we can replace rule (ρ) by the simpler rule:

$$\langle a{=}N\rangle \mid (a a_1 \dots a_n) \ \to\ \langle a{=}N\rangle \mid (N a_1 \dots a_n) \qquad (1)$$

Roughly speaking, we have transformed our functional AM to a chemical AM, or CHAM, in the style of [5]. The most notable improvement is the possibility to compose multiple configurations and, for example, to define configurations with multiple declarations for the same name. Indeed this introduces the possibility of non-deterministic transitions, such as:

$$(\langle a{=}N_1\rangle \mid \langle a{=}N_2\rangle \mid a) \ \to\ (\langle a{=}N_1\rangle \mid \langle a{=}N_2\rangle \mid N_1) \ \text{ or } \ (\langle a{=}N_1\rangle \mid \langle a{=}N_2\rangle \mid N_2)$$

Adding Consumable Resources. Another improvement to our concurrent AM is the addition of a new kind of declaration that is discarded after a communication. We denote $\langle a \Leftarrow P \rangle$ this declaration, and we add the following communication rule:

$$\langle a \Leftarrow N \rangle \mid (a a_1 \dots a_n) \to (N a_1 \dots a_n) \qquad (2)$$

Intuitively, the declaration $\langle a \Leftarrow N \rangle$ allows us to control explicitly the number of accesses to the resource named a, like the input operator in $\boldsymbol{\pi}$. In particular, the declaration $\langle a = N \rangle$ can be interpreted as an infinite parallel composition of "one-shot" declarations $\langle a \Leftarrow N \rangle$. The idea to introduce a consumable kind of resource is not new, and can be found, for example, in the λ-calculus with resources [10, 7], introduced by Boudol to capture the "evaluation blocking" phenomena that are peculiar to concurrent executions. In the encoding of concurrent objects in $\boldsymbol{\pi}^\star$, we will see that objects also appear as a particular kind of declarations

Adding restriction. A mechanism peculiar to the π-calculus, is the possibility to dynamically create fresh names. This can be added very easily in our chemical AM by adding the π-calculus restriction operator, $(\nu a)\mathcal{K}$, together with new reduction rules.

$$\begin{array}{rcll}
(MN)\, a_1 \dots a_n & \to & (\nu a)(\langle a = N \rangle \mid M a a_1 \dots a_n) & (a \text{ fresh name}) \\
\langle a_2 = N \rangle \mid (\nu a_1)\mathcal{K} & \leftrightarrows & (\nu a_1)(\langle a_2 = N \rangle \mid \mathcal{K}) & (a_1 \neq a_2) \\
\mathcal{K} \to \mathcal{K}' & \Rightarrow & (\nu a)\mathcal{K} \to (\nu a)\mathcal{K}' &
\end{array}$$

With restriction, we can for example define the internal choice operator $N \oplus M$ as the term $(\nu a)(\langle a = N \rangle \mid \langle a = M \rangle \mid a)$.

2.2 The Calculus

The blue calculus is the calculus obtained from the chemical AM defined in the previous section, in the same way that the join calculus is derived from the reflexive CHAM defined in [19]. In particular the blue calculus directly extends both the λ-calculus and the π-calculus. Like in the π-calculus, we have non-determism and parallelism, as well as restriction and a communication model based on name passing. Nonetheless, there are also differences between these two calculi. For example, it is possible to directly define higher-order functions in $\boldsymbol{\pi}^\star$.

To define the terms of the blue calculus, we assume given a denumerable set $\mathcal{N}$ of names, ranged over by $a, b, \dots$, and we distinguish three kinds of names: (λ)-variables $x, y, \dots \in \mathcal{V}$, that are bound by the abstraction operator $((\lambda x)P)$, references $u, v, \dots \in \mathcal{R}$, that are bound by restrictions $((\nu u)P)$, and labels $k, l, \dots \in \mathcal{L}$, used to build records. The syntax of $\boldsymbol{\pi}^\star$ is given by the following grammar:

$$\begin{array}{rcll}
P, Q & ::= & a \mid (\lambda x)P \mid (Pa) \mid & \lambda\text{-calculus primitives} \\
 & & (P \mid Q) \mid (\nu u)P \mid & \pi\text{-calculus primitives} \\
 & & \langle u \Leftarrow P \rangle \mid \langle u = P \rangle \mid & \text{declarations} \\
 & & (P \cdot l) \mid [\,] \mid [\, P \,,\, l = Q \,] & \text{records}
\end{array}$$

It is important to notice that, in the definition of $\boldsymbol{\pi}^\star$, we enforce a restricted usage of names with respect to their kinds. Indeed, we only allow declaration on references, for example the term $(\lambda x)\langle x \Leftarrow P \rangle$ is not valid, and we only allow abstraction on variables. Therefore, with these syntactic constraints, it is impossible to create a declaration on a received name. This is comparable, in the π-calculus, to the restriction that only the output-capability of names can be transmitted [31]. A "mobile calculus" with this property is called *local* and it is, for example, the situation of the join calculus where the inputs, the analogue of declarations in $\boldsymbol{\pi}^\star$, are always restricted.

The choice of a local version of the blue calculus differs from the original presentation of $\boldsymbol{\pi}^\star$ [8], but there are also other differences, like for example the use of a "left-commutative" parallel composition operator, a choice that we have motivated in the previous section. Another difference is the extension with record primitives, that are used to interpret objects. The basic record constructor is $[\,P\,,\,l = Q\,]$ that, informally, extends P if the field l is not already present, or updates it. This incremental definition of records is visible in the two reduction rules (red sel) and (red over), given in Table 2. It is possible to give our interpretation of **concς** using only updateable (and not extensible) records, but we prefer to keep the system that we have already successfully used on other occasions [9, 13].

The symbol $\tilde{a}$ denotes the tuple of names $(a_1, \ldots, a_n)$ and, as usual, we abbreviate a sequence of abstractions $(\lambda x_1) \ldots (\lambda x_n) P$ into $(\lambda \tilde{x}) P$. The same convention applies for $(\nu \tilde{u}) P$. We also abbreviate a sequence of applications $((P a_1) \ldots) a_n)$ into $(P \tilde{a})$, and a sequence of extension $[\,[\,[\,]\,, l_1 = P_1\,]\,, \ldots, l_n = P_n\,]$, with the labels l_i all distinct, into $[\,l_i = P_i^{\,i \in [1..n]}\,]$. We denote by $\mathbf{fn}(P)$ the set of free names of P. The operation of substituting the name b for a in P is denoted $P\{^b/_a\}$. It may require converting bound names to avoid capturing b. We will in fact implicitly consider terms up to α-conversion, that is, up to renaming of bound names.

$((P \mid Q) \mid R) \equiv (P \mid (Q \mid R))$		associativity
$((P \mid Q) \mid R) \equiv ((Q \mid P) \mid R)$		semi-commutativity
$(\nu u) P \mid Q \equiv (\nu u)(P \mid Q)$	$(u \notin \mathbf{fn}(Q))$	scope extrusion
$Q \mid (\nu u) P \equiv (\nu u)(Q \mid P)$	$(u \notin \mathbf{fn}(Q))$	
$(\nu u)(\nu v) P \equiv (\nu v)(\nu u) P$		
$(P \mid Q) a \equiv P \mid (Q a)$		distributivity (on parallel composition)
$(P \mid Q) \cdot l \equiv P \mid (Q \cdot l)$		
$((\nu u) P) a \equiv (\nu u)(P a)$	$(a \neq u)$	distributivity (on restriction)
$((\nu u) P) \cdot l \equiv (\nu u)(P \cdot l)$		

Table 1: Structural Congruence

The operational semantics of $\boldsymbol{\pi}^\star$ is given in Tables 1 and 2. The reduction relation is given in a chemical style, that is reduction is defined using two relations: (1) *structural congruence* $\equiv$, equivalent to the relation $\leftrightarrows$ of Sect. 2.1, that is used to rearrange terms. It is the smallest congruence that verifies the axiom given in Table 1; and (2) *reduction* $\rightarrow$, that represents real computation steps. In the definition of the reduction relation, we use the notion of *evaluation contexts* $\mathbf{E}, \mathbf{F}, \ldots$, that are contexts inside which reduction can freely occur. The set of evaluation contexts $\mathcal{E}$, is generated by the following grammar,

$$\mathbf{E} ::= [.] \;\big|\; (\mathbf{E} a) \;\big|\; (\mathbf{E} \cdot l) \;\big|\; (\mathbf{E} \mid P) \;\big|\; (P \mid \mathbf{E}) \;\big|\; (\nu u) \mathbf{E} \tag{3}$$

where the symbol [.] denotes the empty context. The operation $\mathbf{E}[P]$, of filling the hole of a context, is defined in the obvious way.

$$\frac{P \to P' \qquad \mathbf{E} \in \mathcal{E}}{\mathbf{E}[P] \to \mathbf{E}[P']} \text{ (red context)} \qquad \frac{P \to P' \qquad P \equiv Q}{Q \to P'} \text{ (red struct)}$$

$$\frac{}{((\lambda u)P)v \to P\{^{v}/_{u}\}} \text{ (red beta)}$$

$$\frac{}{\langle u \Leftarrow P \rangle \mid (u a_1 \ldots a_n) \to (P a_1 \ldots a_n)} \text{ (red decl)}$$

$$\frac{}{\langle u = P \rangle \mid (u a_1 \ldots a_n) \to \langle u = P \rangle \mid (P a_1 \ldots a_n)} \text{ (red mdecl)}$$

$$\frac{}{[\, P \,,\, l = Q \,] \cdot l \to Q} \text{ (red sel)} \qquad \frac{k \neq l}{[\, P \,,\, l = Q \,] \cdot k \to P \cdot k} \text{ (red over)}$$

Table 2: Reduction

We can divide the basic reduction steps in two. The communication part, that corresponds to the rules (red decl) and (red mdecl), is responsible of the the non-deterministic reduction. The second part correspond to β-reduction and record selection, that is to the rules (red beta), (red sel) and (red beta). This is the functional part of the reduction relation and we denote $\to_{(\beta)}$ the relation obtained from these rules plus (red context) and (red struct). We denote $\to_{(\rho)}$ the reduction relation that correspond to the rule (red decl), (red mdecl), (red context) and (red struct). It is easy to show that if $P \to P'$, then either $P \to_{(\beta)} P'$, or $P \to_{(\rho)} P'$. Moreover in [8], the author shows that $\to_{(\beta)}$, for a slightly different version of $\boldsymbol{\pi}^{\star}$, is Church-Rosser and strongly normalising. This result also holds in our setting, that is:

Theorem 2.1. *If $P \to_{(\beta)} P_1$ and $P \to_{(\beta)} P_2$, then $P_1 \equiv P_2$ or there exists a term P' such that $P_1 \to_{(\beta)} P'$ and $P_2 \to_{(\beta)} P'$. Moreover, for every term P, there exist a term Q such that $P \overset{*}{\to}_{(\beta)} Q$ and Q is β-irreducible, that is $\left\{Q' \;\middle|\; Q \to_{(\beta)} Q'\right\} = \emptyset$.*

2.3 Derived Operators

To simplify the presentation of our encoding and of the type system, we introduce three derived operators, namely:

$$\begin{aligned} \mathbf{def}\, u = P \,\mathbf{in}\, Q &=_{\text{def}} (\nu u)(\langle u = P \rangle \mid Q) \\ \mathbf{set}\, x = P \;\mathbf{in}\, Q &=_{\text{def}} (\nu u)(\langle u \Leftarrow (\lambda x) Q \rangle \mid (P u)) \\ \mathbf{reply}(a) &=_{\text{def}} (\lambda r)(r\ a) \end{aligned}$$

We may interpret the first derived operator, subsequently called a definition, as an explicit substitution of P for the name u in Q. In particular, we can define application (PQ) as $(\mathbf{def}\, u = Q \,\mathbf{in}\, (Pu))$, where $u \notin \mathbf{fn}(P) \cup \mathbf{fn}(Q)$. Note that the name u is recursively bound in a definition, and that it is possible to define a recursion operator $\mathbf{rec}\, u.P$, as a shorthand for $\mathbf{def}\, u = P \,\mathbf{in}\, u$.

The derived operator $(\mathbf{set}\, u = P \;\mathbf{in}\, Q)$, that we call a *linear application* since, in its definition, the declaration $\langle u \Leftarrow (\lambda x) Q \rangle$ can be accessed at most once, is the equivalent of a "call by value" definition. In particular, sequential composition $(P \,;\, Q)$ can be defined as $(\mathbf{set}\, u = P \;\mathbf{in}\, Q)$, for some u not free in Q. The **reply** operator is used in continuation passing style encodings and, for example, to return a value in a linear application, like in (4).

Informally, the **reply** operator is comparable to the use of "synchronous name" in the join calculus [19]. The difference is that, using our notation for higher-order application, it is also possible return a general term:

$$\mathbf{reply}(P) \;=\; (\lambda r)(rP) \;=\; (\mathbf{def}\, u = P\, \mathbf{in}\, (\lambda r)(ru)) \qquad (u, r \notin \mathbf{fn}(P))$$

Another example using the continuation operator is given in (13).

$$\mathbf{set}\, x = \mathbf{reply}(a)\ \mathbf{in}\, Q \;\rightarrow\; (\nu u)(\langle u \Leftarrow (\lambda x)Q \rangle \mid (ua)) \;\overset{*}{\rightarrow}\; (\nu u)(Q\{^{a}/_{x}\}) \tag{4}$$

2.4 Equivalence

We define a relation of observational equivalence between $\boldsymbol{\pi}^\star$-terms ($\approx_b$) used to prove the correctness of our interpretation of **conc**$\boldsymbol{\varsigma}$, *see* Theorem 4.1. This relation, that is a variant of the weak barbed congruence [32], is the biggest bisimulation that preserves simple observations called barbs and that is a congruence [13, 27]. The difference here is that, whereas the observable behaviors considered in CCS and $\boldsymbol{\pi}$ are the visible outputs, we choose instead to observe the presence of values, ie. abstractions, as in the definition of traditional Morris-style equivalences in the λ-calculus [4, ex. 16.5.5]. Informally, the term P is a value (or is observable), denoted $P\downarrow$, if it can react to an application or a selection from the outside world, that is if it belongs to the set generated by the following grammar:

$$V \;::=\; (\lambda x)P \;\big|\; [P\,,\, l = Q] \;\big|\; (P \mid V) \;\big|\; (\nu u)V \tag{5}$$

The weak version of barbs used in the definition of $\approx_b$ is $P \Downarrow$, that is true iff it exists a value V such that $P \overset{*}{\rightarrow} V$.

Definition 2.2 (Barbed Bisimulation). *A relation $\mathcal{S}$ is a* weak barbed simulation *if for each $(P, Q) \in \mathcal{S}$, (1) : whenever $P \rightarrow P'$ then $Q \overset{*}{\rightarrow} Q'$ and $(P', Q') \in \mathcal{S}$; (2) : $P\downarrow$ implies $Q \Downarrow$. $\mathcal{S}$ is a weak barbed bisimulation if $\mathcal{S}$ and $\mathcal{S}^{-1}$ are weak barbed simulation. P and Q are observationally equivalent, written $P \approx_b Q$, iff $(P, Q) \in \mathcal{S}$ for some weak barbed bisimulation $\mathcal{S}$ that is also a congruence.*

The strong barbed congruence $\sim_b$, is defined in the obvious way. In [13], we proved several equational laws for a variant of $\boldsymbol{\pi}^\star$, that are also valid in our setting. In particular we have shown that a definition can be safely distributed and replicated over any contexts, and that β-reduction and "definition fetching" are deterministic computation steps. For example, we have proved that the following relations hold:

$$\mathbf{def}\, u = R\, \mathbf{in}\, (P \mid Q) \sim_b (\mathbf{def}\, u = R\, \mathbf{in}\, P) \mid (\mathbf{def}\, u = R\, \mathbf{in}\, Q) \tag{6}$$

$$P \equiv Q \;\Rightarrow\; P \sim_b Q \tag{7}$$

$$\mathbf{def}\, u = R\, \mathbf{in}\, P \sim_b P \qquad (\text{if } u \notin \mathbf{fn}(P)) \tag{8}$$

$$\mathbf{rec}\, u.R \approx_b \mathbf{def}\, u = R\, \mathbf{in}\, R \tag{9}$$

$$((\lambda x)P)\ a \approx_b P\{^{a}/_{x}\} \tag{10}$$

Two main reasons have motivated our choice to consider barbed congruence for reasonning about blue processes. First, barbed congruence is "*a uniform basis to define equivalences between different process calculi*" [6], and therefore we can establish comparison with results obtained in other calculi. Second, the choice of a relation that is at the same time a congruence and a bisimulation ease proofs of interpretations and term transformations. Nonetheless there is also a drawback, namely that proofs of equational laws for barbed congruence introduce quantification over all possible contexts. To avoid this pitfall, we use a bisimulation for $\boldsymbol{\pi}^\star$, based on a labeled transition system defined in [13], that is finer than barbed congruence. Indeed, we have defined a strong bisimulation relation, denoted $\sim_d$, and a weak bisimulation, $\approx_d$, such that $\sim_d \; \subset \; \sim_b$ and $\approx_d \; \subset \; \approx_b$.

Proofs for labeled bisimulations are simpler than with barbed congruence. Indeed we can use very powerfull proofs methods that help to reduce the size of the relations that we have to consider. For example, in Sect. 6.1, we use proofs up-to strong bisimulation and up-to expansion [37, 38] to prove equational laws on the encoding of **concς** terms.

We will not define the relations $\sim_d$ and $\approx_d$ in this paper, nor the expansion relation ($\gtrsim_d$), that is a dissymetric version of $\approx_d$ such that $\sim_d \; \subset \; \gtrsim_d \; \subset \; \approx_d$. Let us just say that $\sim_d$ verifies the laws (6), (7) and (8) and that, intuitively, $P \gtrsim_d Q$ means that Q has to do more internal transitions than P, or also that Q is less efficient than P. In Sect. 6.1, we will use the fact that if $P \rightarrow_{(\beta)} P'$, then $P \gtrsim_d P'$. We will also use the property that, if $\mathbf{C}$ is a context that does not bind the name u or the free names in R, then:

$$\mathbf{def}\, u = R \,\mathbf{in}\, \mathbf{C}[u] \;\gtrsim_d\; \mathbf{def}\, u = R \,\mathbf{in}\, \mathbf{C}[R] \tag{11}$$

$$(\nu u)(\langle u \Leftarrow P \rangle \mid u) \;\gtrsim_d\; (\nu u)P \tag{12}$$

For example we can use these laws to prove the following relations:

$$\begin{aligned} (\mathbf{reply}(a)\ (\lambda y)R) \;&\gtrsim_d\; \mathbf{def}\, u = (\lambda y)R \,\mathbf{in}\, (ua) \qquad (u \notin \mathbf{fn}(R)) \\ &\gtrsim_d\; \mathbf{def}\, u = (\lambda y)R \,\mathbf{in}\, (R\{^a/_y\}) \\ &\sim_d\; R\{^a/_y\} \end{aligned} \tag{13}$$

2.5 The reference cell

We study a simple example, the mutable cell, that can be interpreted as the paradigmatic example of object. Indeed, in our intuition, the identity of an object is a reference at which the object state can be fetched, its state is a record of methods (as in the classical recursive records semantic [12]) and encapsulation is naturally implemented using declaration and restriction. Let $\mathsf{R}(s,x)$ denotes the record:

$$\mathsf{R}(s,x) \;=_{\text{def}}\; [\,get = (sx \mid \mathbf{reply}(x)),\; put = s\,] \tag{14}$$

The cell of name p, initialized with the value a_0, is the term defined by:

$$\mathsf{Cell}(a_0) \;=_{\text{def}}\; \mathbf{def}\, s = (\lambda x)\langle p \Leftarrow \mathsf{R}(s,x)\rangle \,\mathbf{in}\, \langle p \Leftarrow \mathsf{R}(s,a_0)\rangle \tag{15}$$

It is easy to show that the term $\mathsf{Cell}(a_0)$ is the result of applying the recursive definition $\mathbf{rec}\, s.((\lambda x)\langle p \Leftarrow \mathsf{R}(s,x)\rangle)$ to the name a_0, that is we have:

$$\mathbf{rec}\, s.((\lambda x)\langle p \Leftarrow \mathsf{R}(s,x)\rangle)\ a_0 \overset{*}{\rightarrow} \mathsf{Cell}(a_0) \tag{16}$$

Therefore the cell can be informally divided into two components. An active part, the declaration $\langle p \Leftarrow \mathsf{R}(s,x)\rangle$, that can interact with other processes in parallel. A passive part, the recursive definition on the name s, that is used to recreate a fresh definition each time the cell is accessed. The cell object can be invoked in its two method, *get* and *put*, that represent, respectively, a read and and a write operation on the cell. For example we have:

$$\begin{array}{rcl}
\mathsf{Cell}(a_0) \mid (p \cdot get\ (\lambda y) R) & \equiv & \mathbf{def}\, s = (\lambda x)\langle p \Leftarrow \mathsf{R}(s,x)\rangle\, \mathbf{in}\, (\langle p \Leftarrow \mathsf{R}(s,a_0)\rangle \mid p \cdot get\ (\lambda y) R) \\
 & \rightarrow & \mathbf{def}\, s = (\lambda x)\langle p \Leftarrow \mathsf{R}(s,x)\rangle\, \mathbf{in}\, (\mathsf{R}(s,a_0) \cdot get\ (\lambda y) R) \\
 & \rightarrow & \mathbf{def}\, s = (\lambda x)\langle p \Leftarrow \mathsf{R}(s,x)\rangle\, \mathbf{in}\, (s a_0 \mid \mathbf{reply}(a_0)\ (\lambda y) R) \\
 & \approx_b & \mathsf{Cell}(a_0) \mid R\{a_0/y\}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{Cell}(a_0) \mid (p \cdot put\ a_1) & \overset{*}{\rightarrow} & \mathbf{def}\, s = (\lambda x)\langle p \Leftarrow \mathsf{R}(s,x)\rangle\, \mathbf{in}\, (s a_1) \\
 & \overset{*}{\rightarrow} & \mathsf{Cell}(a_1)
\end{array}$$

It is interesting to notice the linear use of the reference p in $\mathsf{Cell}(a_0)$. If the cell is invoked, we consume the unique declaration on p and a unique message $(p\ a)$, acting like a lock, is freed in the evaluation process, which, in turn, frees a single declaration on p. Therefore, we have the invariant that there is exactly one resource available at address p, and that this resource keeps the last value passed in a "call like": $(p \cdot put)$ call.

3 The Source Calculus

The calculus **concς** is a calculus based on the notion of naming, obtained by extending the imperative **ς**-calculus with operators taken from **π**. In particular, parallel composition and restriction are two operators of **concς**. As the imperative **ς**-calculus, it also provides an operator to clone an object, and a call-by-value definition operator: $\mathbf{let}\, x = a\, \mathbf{in}\, b$.

The syntax of **concς** is given in Table 3. The basic constructor is the denomination: $(p \mapsto [\, l_i = \varsigma(x_i) b_j{}^{i \in [1..n]}\,])$, that, informally, adds a name to a **ς**-object and acts like a special kind of declaration. As in our version of the blue calculus, the parallel composition operator is dissymetrical. Rule (struct par comm) allows commutations only at the left of a parallel composition and therefore it is possible to define the result part of a process as the rightmost term (*see* the reduction rule (gh red let result)). The intuition is that the $\Rsh$ operator acts like the *fork* instruction found in some programming language with threads: executing $(a \Rsh b)$ amounts to create a fresh thread a, that has only side effects; the result of evaluating $(a \Rsh b)$ is the result of b.

As for the π and blue calculi, the operational semantics is given in a chemical style. Structural equivalence, denoted $\equiv$, is the smallest congruence that verifies the axioms given in Table 4. The reduction relation $\rightarrow$, is given in Table 5.

All the rules in Table 4 are direct equivalent of blue calculus structural rules (*see* Table 1), apart rules (struct par let) and (struct res let), that allow interaction between a, in $\mathbf{let}\, x = a\, \mathbf{in}\, b$, and other terms in parallel. We have decided to omit a structural equivalence rule for **concς** that is given in [21], that is the rule (struct let assoc) below:

$$\frac{y \notin \mathbf{fn}(c)}{\mathbf{let}\, x = (\mathbf{let}\, y = a\, \mathbf{in}\, b)\, \mathbf{in}\, c \equiv \mathbf{let}\, y = a\, \mathbf{in}\, (\mathbf{let}\, x = f\, \mathbf{in}\, c)}\ \text{(struct let assoc)} \qquad (17)$$

This rule is part of the original definition of **concς**, but it is used only to simplify the definition of normal forms and therefore we decide to not consider this rule. This choice is motivated by two other remarks. First, the absence of this rule does not affect reduction, in the sense that Hankin conjectures the following property [24]:

- if $a \rightarrow_{\mathrm{LA}} b$, then there exists a term b' such that $a \rightarrow b' \equiv b$;

results:	u, v	$::=$	x	variable
		$\mid$	p	name
denotations :	d	$::=$	$[\,l_i = \varsigma(x_i)b_j{}^{i\in[1..n]}\,]$	
terms:	a, b, c	$::=$	u	result
		$\mid$	$(p \mapsto d)$	denomination
		$\mid$	$u{\cdot}l$	method invocation
		$\mid$	$u{\cdot}l \Leftarrow \varsigma(x)b$	method update
		$\mid$	$\mathbf{clone}(u)$	cloning
		$\mid$	$\mathbf{let}\, x = a \,\mathbf{in}\, b$	let
		$\mid$	$(a \upharpoonright b)$	parallel composition
		$\mid$	$(\nu p)a$	restriction

Table 3: Syntax of **conc**$\boldsymbol{\varsigma}$

$$\frac{}{(a \upharpoonright b) \upharpoonright c \equiv a \upharpoonright (b \upharpoonright c)}\ \text{(struct par assoc)} \qquad \frac{}{(a \upharpoonright b) \upharpoonright c \equiv (b \upharpoonright a) \upharpoonright c}\ \text{(struct par comm)}$$

$$\frac{}{(\nu p)(\nu q)a \equiv (\nu q)(\nu p)a}\ \text{(struct res res)} \qquad \frac{p \notin \mathbf{fn}(a)}{(\nu p)(a \upharpoonright b) \equiv (a \upharpoonright (\nu p)b)}\ \text{(struct par)}$$

$$\frac{p \notin \mathbf{fn}(b)}{(\nu p)(a \upharpoonright b) \equiv ((\nu p)a \upharpoonright b)}\ \text{(struct par)} \qquad \frac{p \notin \mathbf{fn}(b)}{(\nu p)(\mathbf{let}\, x = a \,\mathbf{in}\, b) \equiv \mathbf{let}\, x = (\nu p)a \,\mathbf{in}\, b}\ \text{(struct res let)}$$

$$\frac{}{a \upharpoonright \mathbf{let}\, x = b \,\mathbf{in}\, c \equiv \mathbf{let}\, x = (a \upharpoonright b) \,\mathbf{in}\, c}\ \text{(struct par let)}$$

Table 4: Structural Equivalence in **conc**$\boldsymbol{\varsigma}$

– if $a \to b$, then $a \to_{\mathrm{LA}} b$.

Second, it is clear that the set of couples $(\mathbf{let}\, x = (\mathbf{let}\, y = a \,\mathbf{in}\, b) \,\mathbf{in}\, c, \mathbf{let}\, y = a \,\mathbf{in}\, (\mathbf{let}\, x = f \,\mathbf{in}\, c))$, such that $y \notin \mathbf{fn}(c)$, is a strong bisimulation up-to restriction and parallel composition [38].

Let d be the denotation $d =_{\mathrm{def}} [\, l_i = \varsigma(x_i)b_i{}^{i \in [1..n]}\,]$

$$\frac{a \to a' \qquad a \equiv b}{b \to a'} \text{ (gh red struct)} \qquad \frac{a \to a'}{\mathbf{let}\, x = a \,\mathbf{in}\, b \to \mathbf{let}\, x = a' \,\mathbf{in}\, b} \text{ (gh red let)}$$

$$\frac{a \to a'}{(\nu p)a \to (\nu p)a'} \text{ (gh red res)} \qquad \frac{a \to a'}{a \Rsh b \to a' \Rsh b} \text{ (gh red par 1)} \qquad \frac{b \to b'}{a \Rsh b \to a \Rsh b'} \text{ (gh red par 2)}$$

$$\frac{j \in [1..n]}{(p \mapsto d) \Rsh p{\cdot}l_j \;\to\; (p \mapsto d) \Rsh b_j\{p/x_j\}} \text{ (gh red select)}$$

$$\frac{d' = [\, l_i = \varsigma(x_i)b_i{}^{i \in [1..n], i \neq j}, l_j = \varsigma(x)b \,] \qquad j \in [1..n]}{(p \mapsto d) \Rsh (p{\cdot}l_j \Leftarrow \varsigma(x)b) \to (p \mapsto d') \Rsh p} \text{ (gh red updt)}$$

$$\frac{q \notin \mathbf{fn}(d)}{(p \mapsto d) \Rsh \mathbf{clone}(p) \to (p \mapsto d) \Rsh (\nu q)((q \mapsto d) \Rsh q)} \text{ (gh red clone)}$$

$$\frac{}{\mathbf{let}\, x = p \,\mathbf{in}\, b \to b\{p/x\}} \text{ (gh red let result)}$$

Table 5: Reduction in **conc$\boldsymbol{\varsigma}$**

4 Coding concς in the Blue Calculus

In this section, we interpret **conc$\boldsymbol{\varsigma}$** in the blue calculus and we prove a full-abstraction result between the two calculi. In the definition of the **conc$\boldsymbol{\varsigma}$** encoding, we define a set of notations that allows us to interpret the blue calculus as an object calculus.

We suppose that the denomination names are also references, that is $p, q \ldots$ are in $\mathcal{R}$. The interpretation of a denomination is inspired by the reference cell of Sect. 2.5. Indeed, a denomination $(p \mapsto d)$, where d is the denotation $[\, l_i = \varsigma(x_i)b_i{}^{i \in [1..n]}]$, can be interpreted has a cell with n values: $((\lambda x_i)[\![b_i]\!])_{i \in [1..n]}$. Therefore it is a declaration that encapsulates a record with $2n$ fields. The "access field" get_{l_i} (with $i \in [1..n]$), that allows to invoke method l_i, and the field put_{l_i} that allows to modify this method. We also add a field named *clone* that, when selected, creates a fresh cell with a copy of its current state. Schematically, we use the "*split-method*" technique of [2]. Let $\mathsf{R}(p, s, \tilde{x}, c)$ be the following record.

$$\mathsf{R}(p, s, \tilde{x}, c) =_{\mathrm{def}} \begin{bmatrix} \ldots & {}^{i \in [1..n]} \\ get_{l_i} = (s\tilde{x} \mid x_i p), & \\ put_{l_i} = (\lambda y)(s x_1 \ldots y \ldots x_n \mid \mathbf{reply}(p)), & \\ \ldots & \\ clone = (s\tilde{x} \mid c\tilde{x}) & \end{bmatrix} \tag{18}$$

It is interesting to compare the record $\mathsf{R}(p,s,\tilde{x})$ with the record $\mathsf{R}(s,x)$ of (14). For example, the result of selecting the field get_{l_j} in $\mathsf{R}(p,s,\tilde{x},c)$, is to apply the j^{th} memorized value to the name of the denomination. Informally, this allow us to interpret method invocation, that is, if the names in $\tilde{x}$ are binded to the values $((\lambda x_i)[\![b_i]\!])_{i\in[1..n]}$, we have: $(\langle p\Leftarrow\mathsf{R}(p,s,\tilde{x},c)\rangle \mid (p\cdot get_{l_j})) \overset{*}{\rightarrow} \cdots \mid ((\lambda x_j)[\![b_j]\!])\ p$.

As in the example of the reference cell, the record $\mathsf{R}(p,s,\tilde{x},c)$ is encapsulated in a recursive definition that "linearly manages" a declaration of the name p. We define a notation for this definition, namely:

$$\mathsf{Fobj}(p,\tilde{x},c) =_{\text{def}} \mathbf{def}\ s = (\lambda\tilde{y})\langle p\Leftarrow\mathsf{R}(p,s,\tilde{y},c)\rangle\ \mathbf{in}\ \langle p\Leftarrow\mathsf{R}(p,s,\tilde{x},c)\rangle \tag{19}$$

We denote $\langle p\mapsto\{\, l_i = (\lambda x_i)P_i{}^{i\in[1..n]}\,\}\rangle$ the process that we obtain by binding the name c to the function that clone the object, and the names in $\tilde{x}$ to the "premethods" $((\lambda x_i)P_i)_{i\in[1..n]}$. In the remainder of this paper, we will use the symbol L to denote an object body $\{\, l_i = (\lambda x_i)P_i{}^{i\in[1..n]}\,\}$.

$$\begin{aligned}\langle p\mapsto L\rangle \ =_{\text{def}}\ & \mathbf{def}\ c = (\lambda\tilde{x})(\nu q)(\mathsf{Fobj}(q,\tilde{x},c) \mid \mathbf{reply}(q))\\ & \mathbf{in}\ \mathbf{def}\ u_1 = (\lambda x_1)P_1,\ldots,u_n = (\lambda x_n)P_n\\ & \quad \mathbf{in}\ \mathsf{Fobj}(p,\tilde{u},c)\end{aligned} \tag{20}$$

The interpretation of **conc**$\boldsymbol{\varsigma}$ is given in Table 6. We can simplify this interpretation a step further by defining a shorthand for method select, method update and for cloning, namely:

$$\mathbf{clone}(P) = (P\ \cdot clone) \qquad (P \Leftarrow l) = (P\ \cdot get_l) \qquad (P\cdot l \Leftarrow (\lambda x)Q) = (P\ \cdot put_l\ (\lambda x)Q)$$

This allows us to consider that **conc**$\boldsymbol{\varsigma}$ is embedded in the blue calculus. More precisely, we embed an "higher-order" version of **conc**$\boldsymbol{\varsigma}$. Indeed, with these derived operators, it is possible to apply an object operator to a term that is not a result, such as in $\mathbf{clone}(P \mid Q)$ for example, or to define a selector method: $(\lambda x)(x \Leftarrow l)$, two terms that are not directly expressible in **conc**$\boldsymbol{\varsigma}$.

In the remainder of this section, we prove that there is an operational correspondence between **conc**$\boldsymbol{\varsigma}$ and the blue calculus. To simplify the proofs, we define the evaluation context $\mathbf{E_p}$ such that:

$$\mathbf{E_p} =_{\text{def}} \left(\begin{array}{l}\mathbf{def}\ c = (\lambda\tilde{x})(\nu q)(\mathsf{Fobj}(q,\tilde{x},c) \mid \mathbf{reply}(q)),\\ \mathbf{in} \left(\begin{array}{l}\mathbf{def}\ u_1 = (\lambda x_1)P_1,\ldots,u_n = (\lambda x_n)P_n\\ \mathbf{in}\ (\mathbf{def}\ s = (\lambda\tilde{x})\langle p\Leftarrow\mathsf{R}(p,s,\tilde{x},c)\rangle\ \mathbf{in}\ [.])\end{array}\right)\end{array}\right) \tag{21}$$

It is easy to show that $\langle p\mapsto L\rangle$ is equal to $\mathbf{E_p}[\langle p\Leftarrow\mathsf{R}(p,s,\tilde{x},c)\rangle]$ and that the term $\mathbf{E_p}[s\tilde{u}]$ reduces to $\langle p\mapsto L\rangle$. More precisely it is obtainded using β-reduction and definition fetching, and therefore, using the result on the expansion relation gievn in Sect. 2.4, it follows that: $\mathbf{E_p}[s\tilde{u}] \gtrsim_d \langle p\mapsto L\rangle$.

Theorem 4.1 (Completeness). *If $a \equiv b$, then $[\![a]\!] \equiv [\![b]\!]$. If $a \rightarrow a'$, then $[\![a]\!] \overset{*}{\rightarrow} \sim_b [\![a']\!]$.*

Proof. The proof of the first implication is by induction on the inference of $a \equiv b$. We only give the cases for the rules (struct res let) and (struct par let), the only rules that have no direct counterpart in $\boldsymbol{\pi}^\star$.

$$[\![(p \mapsto [\, l_i = \varsigma(x_i)b_i{}^{i\in[1..n]}\,])]\!] = \langle p \mapsto \{\, l_i = (\lambda x_i)[\![b_i]\!]^{i\in[1..n]}\,\}\rangle$$

$$[\![u]\!] = \mathbf{reply}(u) \qquad [\![u{\cdot}l]\!] = u\ \cdot get_l \qquad [\![u{\cdot}l \Leftarrow \varsigma(x)b]\!] = u\ \cdot put_l\ (\lambda x)[\![b]\!] \qquad [\![\mathbf{clone}(u)]\!] = u\ \cdot clone$$

$$[\![\mathbf{let}\, x = a\,\mathbf{in}\, b]\!] = \mathbf{set}\, x = [\![a]\!]\ \mathbf{in}\ [\![b]\!] \qquad [\![a \Rsh b]\!] = ([\![a]\!] \mid [\![b]\!]) \qquad [\![(\nu p)a]\!] = (\nu p)[\![a]\!]$$

Table 6: Interpretation of **concς**

(struct res let) Let p be a name that is not free in b, and let u be a fresh name, we have:

$$\begin{aligned}[\![(\nu p)(\mathbf{let}\, x = a\,\mathbf{in}\, b)]\!] &= (\nu p)((\nu u)(\langle u{\Leftarrow}(\lambda x)[\![b]\!]\rangle \mid [\![a]\!]\, u))\\ &\equiv (\nu u)(\langle u{\Leftarrow}(\lambda x)[\![b]\!]\rangle \mid ((\nu p)[\![a]\!])\, u)\\ &= [\![\mathbf{let}\, x = (\nu p)a\,\mathbf{in}\, b]\!]\end{aligned}$$

(struct par let) let u be a fresh name, we have:

$$\begin{aligned}[\![a \Rsh \mathbf{let}\, x = b\,\mathbf{in}\, c]\!] &= [\![a]\!] \mid (\nu u)(\langle u{\Leftarrow}(\lambda x)[\![c]\!]\rangle \mid [\![b]\!]\, u)\\ &\equiv (\nu u)(\langle u{\Leftarrow}(\lambda x)[\![c]\!]\rangle \mid ([\![a]\!] \mid [\![b]\!])\, u)\\ &= [\![\mathbf{let}\, x = (a \Rsh b)\,\mathbf{in}\, c]\!]\end{aligned}$$

Proof of the second property is by induction on the inference of $a \to a'$. Let L be the object body $\{\, l_i = (\lambda x_i)P_i^{i\in[1..n]}\,\}$.

(gh red select) Let P be the term $(\langle p \mapsto L\rangle \mid p \Leftarrow l_j)$. We prove that if $j \in [1..n]$, then $P \xrightarrow{*} (\langle p \mapsto L\rangle \mid P_j\{p/x_j\})$. Using the notation given in (21), we have that $P = \mathbf{E_p}[\langle p{\Leftarrow}\mathsf{R}(p,s,\tilde{u},c)\rangle] \mid p\ \cdot get_{l_j}$. Moreover we have already showed that $\mathbf{E_p}[s\tilde{u}]$ reduces to $\langle p \mapsto L\rangle$. Therefore we have that:

$$\begin{aligned}P &\equiv \mathbf{E_p}[\langle p{\Leftarrow}\mathsf{R}(p,s,\tilde{u},c)\rangle \mid p\cdot get_{l_j}]\\ &\to_{(\rho)} \mathbf{E_p}[\mathsf{R}(p,s,\tilde{u},c)\cdot get_{l_j}]\\ &\to_{(\beta)} \mathbf{E_p}[s\tilde{u} \mid u_j p]\\ &\to_{(\rho)} \mathbf{E_p}[s\tilde{u} \mid ((\lambda x_j)P_j)p]\\ &\to_{(\beta)} \mathbf{E_p}[s\tilde{u} \mid P_j\{p/x_j\}]\\ &\equiv \mathbf{E_p}[s\tilde{u}] \mid P_j\{p/x_j\}\\ &\xrightarrow{*} \langle p \mapsto L\rangle \mid P_j\{p/x_j\}\end{aligned}$$

(gh red updt) Let j be a natural number in the interval $[1..n]$, L' be the body $\{\, l_i = (\lambda x_i)P_i^{i\in[1..n], i\neq j}, l_j = (\lambda x)Q\,\}$, and P be the term $(\langle p \mapsto L\rangle \mid (p{\cdot}l_j \Leftarrow (\lambda x)Q))$. We prove that $P \xrightarrow{*} \sim_b \langle p \mapsto L'\rangle \mid \mathbf{reply}(p)$.

$$\begin{aligned}P &\equiv \mathbf{E_p}[\langle p{\Leftarrow}\mathsf{R}(p,s,\tilde{u},c)\rangle \mid p\cdot put_{l_j}\ (\lambda x)Q]\\ &\to_{(\rho)} \mathbf{E_p}[\mathsf{R}(p,s,\tilde{u},c)\cdot put_{l_j}\ (\lambda x)Q]\\ &\to_{(\beta)} \mathbf{E_p}[\mathbf{def}\, y = (\lambda x)Q\,\mathbf{in}\,((\lambda y)(su_1\ldots y\ldots u_n \mid \mathbf{reply}(p)))y]\\ &\to_{(\beta)} \mathbf{E_p}[\mathbf{def}\, y = (\lambda x)Q\,\mathbf{in}\,(su_1\ldots y\ldots u_n \mid \mathbf{reply}(p))]\\ &\sim_b \langle p \mapsto L'\rangle \mid \mathbf{reply}(p)\end{aligned}$$

In the last relation, we use the strong barbed congruence $\sim_b$ to garbage collect the definition $(u_j = (\lambda x_j)P_j)$ from the context $\mathbf{E_p}$, *see* (8) in Sect. 2.4;

(gh red clone) let q be a fresh name, and let P be the term $(\langle p \mapsto L\rangle \mid \mathbf{clone}(p))$. We prove that $P \overset{*}{\to} \sim_b \langle p \mapsto L\rangle \mid (\nu q)(\langle q \mapsto L\rangle \mid \mathbf{reply}(q))$.

$$\begin{array}{rcl} P & \equiv & \mathbf{E_p}[\langle p \Leftarrow \mathsf{R}(p, s, \tilde{u}, c)\rangle \mid p \cdot \mathit{clone}] \\ & \to_{(\rho)} & \mathbf{E_p}[\mathsf{R}(p, s, \tilde{u}, c) \cdot \mathit{clone}] \\ & \to_{(\beta)} & \mathbf{E_p}[s\tilde{u} \mid c\tilde{u}] \\ & \to_{(\rho)} & \mathbf{E_p}[s\tilde{u} \mid ((\lambda\tilde{x})(\nu q)(\mathsf{Fobj}(q, \tilde{x}, c) \mid \mathbf{reply}(q)))\tilde{u}] \\ & \overset{*}{\to}_{(\beta)} & \mathbf{E_p}[s\tilde{u} \mid (\nu q)(\mathsf{Fobj}(q, \tilde{u}, c) \mid \mathbf{reply}(q))] \\ & \sim_b & \mathbf{E_p}[s\tilde{u}] \mid (\nu q)\mathbf{E_p}[(\mathsf{Fobj}(q, \tilde{u}, c) \mid \mathbf{reply}(q))] \\ & \overset{*}{\to} & \langle p \mapsto L\rangle \mid (\nu q)(\langle q \mapsto L\rangle \mid \mathbf{reply}(q)) \end{array}$$

in the penultimate relation, we use the strong barbed congruence relation to duplicate and distribute the definitions on c and the names of $\tilde{u}$, *see* (6);

(gh red let) the hypothesis is that $a \to a'$, therefore, using the induction hypothesis it follows that $\llbracket a\rrbracket \overset{*}{\to} \sim_b \llbracket a'\rrbracket$. Let u be a fresh name, we use the fact that $(\nu u)(\langle u \Leftarrow (\lambda x)\llbracket b\rrbracket\rangle \mid [.]\, u)$ is an evaluation context and that $\sim_b$ is a congruence to show that:

$$\begin{array}{rcl} \llbracket \mathbf{let}\, x = a \,\mathbf{in}\, b\rrbracket & = & (\nu u)(\langle u \Leftarrow (\lambda x)\llbracket b\rrbracket\rangle \mid \llbracket a\rrbracket\, u) \\ & \overset{*}{\to} \sim_b & (\nu u)(\langle u \Leftarrow (\lambda x)\llbracket b\rrbracket\rangle \mid \llbracket a'\rrbracket\, u) \\ & = & \llbracket \mathbf{let}\, x = a' \,\mathbf{in}\, b\rrbracket \end{array}$$

(gh red let result) let u be a fresh name, therefore:

$$\begin{array}{rcl} \llbracket \mathbf{let}\, x = p \,\mathbf{in}\, b\rrbracket & = & (\nu u)(\langle u \Leftarrow (\lambda x)\llbracket b\rrbracket\rangle \mid \mathbf{reply}(p)\, u) \\ & \to & (\nu u)(\langle u \Leftarrow (\lambda x)\llbracket b\rrbracket\rangle \mid u\, p) \\ & \overset{*}{\to} & (\nu u)\llbracket b\rrbracket\{p/x\} \\ & \equiv & \llbracket b\{p/x\}\rrbracket \end{array}$$

□

We have proved that the interpretation $\llbracket . \rrbracket$ is complete, that is $\boldsymbol{\pi}^\star$ can simulate reductions in **concς**, we prove that it is also sound. Before to prove this property, we prove two intermediary results. The proofs will use the notion of *evaluation context* that, as in $\boldsymbol{\pi}^\star$, are contexts inside which evaluation can occurs freely. In **concς**, evaluation contexts: $\mathbf{e}, \mathbf{f}, \ldots$, are the contexts generated by the following grammar:

$$\mathbf{e} ::= [.] \;\big|\; \mathbf{let}\, x = \mathbf{e} \,\mathbf{in}\, a \;\big|\; (\mathbf{e} \Rsh a) \;\big|\; (a \Rsh \mathbf{e}) \;\big|\; (\nu u)\mathbf{e}$$

We will also use the notion of *let context*: $\mathbf{g}, \mathbf{h}, \ldots$, that are evaluation contexts such that the hole can interact with a **let** construct, in the sense that $(\mathbf{let}\, x = \mathbf{g}[u] \,\mathbf{in}\, a) \to \mathbf{g}[a\{u/x\}]$.

$$\mathbf{g} ::= [.] \;\big|\; (a \Rsh \mathbf{g}) \;\big|\; (\nu u)\mathbf{g}$$

Lemma 4.1. *If $\llbracket a\rrbracket \to_{(\beta)} P$, then there exists a term a' such that $a \to a'$ and $P \overset{*}{\to} \sim_b \llbracket a'\rrbracket$. Let v be a fresh reference, if $(\llbracket a\rrbracket v) \to_{(\beta)} P$, then either there exists a term a' such that $a \to a'$ and $P \overset{*}{\to} \sim_b (\llbracket a'\rrbracket v)$, or there exists a let context $\mathbf{g}$ and a result u such that $a \equiv \mathbf{g}[u]$.*

Proof. The proof is by induction on the size of a.

case $a = u$**:** in this case $\llbracket a\rrbracket$ does not reduces, and we have: $(\llbracket a\rrbracket v) \to_{(\beta)} (uv)$. The result follows from the fact that $[.]$ is a let context;

case $a = (p \mapsto d)$**:** in this case $\llbracket a \rrbracket$ and $(\llbracket a \rrbracket v)$ does not reduce. The proof is similar in the case such that a is a method select, a method update or a cloning;

case $a = (\nu p)b$**:** the result follows from the fact that $\llbracket a \rrbracket = (\nu p)\llbracket b \rrbracket$ and that $(\nu p)\mathbf{g}$ is a let context;

case $a = (\mathbf{let}\, x = b \,\mathbf{in}\, c)$**:** we recall that $\llbracket a \rrbracket$ is equal to $(\nu v)(\langle v {\Leftarrow} (\lambda x)\llbracket c \rrbracket\rangle \mid \llbracket b \rrbracket v)$. Suppose that $\llbracket a \rrbracket \rightarrow_{(\beta)} P$. It is easy to show that there exits a term Q such that $(\llbracket b \rrbracket v) \rightarrow_{(\beta)} Q$ and $P \equiv (\nu v)(\langle v {\Leftarrow} (\lambda x)\llbracket c \rrbracket\rangle \mid Q)$. Therefore, using the induction hypothesis, there are two possible cases:

- there exists a term b' such that $b \rightarrow b'$ and $Q \overset{*}{\rightarrow} \sim_b (\llbracket b' \rrbracket v)$. In this case we have that $a \rightarrow (\mathbf{let}\, x = b' \,\mathbf{in}\, c)$ and $P \overset{*}{\rightarrow} \sim_b \llbracket \mathbf{let}\, x = b' \,\mathbf{in}\, c \rrbracket$;
- there exists a let context $\mathbf{g}$ and a result u such that $b \equiv \mathbf{g}[u]$, and we have that (modulo renaming of the bound names in b):

$$\begin{aligned} a &\equiv \mathbf{g}[\mathbf{let}\, x = u \,\mathbf{in}\, c] \rightarrow \mathbf{g}[c\{u/x\}] \\ \llbracket a \rrbracket &\rightarrow P \equiv \llbracket \mathbf{g} \rrbracket[(\nu v)(\langle v {\Leftarrow} (\lambda x)\llbracket c \rrbracket\rangle \mid (vu))] \\ &\rightarrow \llbracket \mathbf{g} \rrbracket[(\nu v)(((\lambda x)\llbracket c \rrbracket)\ u)] \\ &\rightarrow \llbracket \mathbf{g} \rrbracket[(\nu v)\llbracket c\{u/x\} \rrbracket] \\ &\sim_b \llbracket \mathbf{g}[c\{u/x\}] \rrbracket \end{aligned}$$

  In the last relation, we use the strong barbed congruence to garbage collect the restriction on the name v (*see* (8) in sect. 2.4).

case $a = (b \Rsh c)$**:** suppose that $\llbracket a \rrbracket \rightarrow_{(\beta)} P$. It is easy to show that there exists a term Q such that $\llbracket b \rrbracket \rightarrow_{(\beta)} Q$ or $\llbracket c \rrbracket \rightarrow_{(\beta)} Q$,and the result follows using the induction hypothesis. In the case such that $(\llbracket a \rrbracket v) \rightarrow_{(\beta)} P$, we use the fact that there must exists a term Q such that $\llbracket b \rrbracket \rightarrow_{(\beta)} Q$ or such that $(\llbracket c \rrbracket v) \rightarrow_{(\beta)} Q$.

□

Let a_p denotes either a method select, a method update or a cloning on the name p, that is $a_p ::= p{\cdot}l \mid p{\cdot}l \Leftarrow \varsigma(y)b \mid \mathbf{clone}(p)$. In the next lemma, we suppose that d denotes the object $[\, l_i = \varsigma(x_i)b_i{}^{i \in [1..n]}\,]$ and that $\mathbf{E_p}$ denotes the following evaluation context (*see* (18), (19) and (21)):

$$\mathbf{E_p} =_{\mathrm{def}} \left(\begin{array}{l} \mathbf{def}\, c = (\lambda \tilde{x})(\nu q)(\mathsf{Fobj}(q, \tilde{x}, c) \mid \mathbf{reply}(q)), \\ \mathbf{in} \left(\begin{array}{l} \mathbf{def}\, u_1 = (\lambda x_1)\llbracket b_1 \rrbracket, \ldots, u_n = (\lambda x_n)\llbracket b_n \rrbracket \\ \mathbf{in}\, (\mathbf{def}\, s = (\lambda \tilde{x})\langle p {\Leftarrow} \mathsf{R}(p, s, \tilde{x}, c)\rangle \,\mathbf{in}\, [.]) \end{array} \right) \end{array} \right)$$

Lemma 4.2. *if $\llbracket a \rrbracket \rightarrow_{(\rho)} P$, then there exists an evaluation context $\mathbf{e}$ and a name p such that $a \equiv \mathbf{e}[(p \mapsto d) \Rsh a_p]$ and the communication comes from p. That is:*

1. *either $a \equiv \mathbf{e}[(p \mapsto d) \Rsh p{\cdot}l]$ and $P \equiv \llbracket \mathbf{e} \rrbracket[\mathbf{E_p}[\mathsf{R}(p, s, \tilde{x}, c) \cdot get_l]]$;*
2. *or $a \equiv \mathbf{e}[(p \mapsto d) \Rsh p{\cdot}l \Leftarrow \varsigma(y)b]$ and $P \equiv \llbracket \mathbf{e} \rrbracket[\mathbf{E_p}[\mathsf{R}(p, s, \tilde{x}, c) \cdot put_l\ (\lambda y)\llbracket b \rrbracket]]$;*
3. *or $a \equiv \mathbf{e}[(p \mapsto d) \Rsh \mathbf{clone}(p)]$ and $P \equiv \llbracket \mathbf{e} \rrbracket[\mathbf{E_p}[\mathsf{R}(p, s, \tilde{x}, c) \cdot clone]]$.*

Proof. The proof is by induction on the size of a. We prove that if $\llbracket a \rrbracket \rightarrow_{(\rho)} P$, then a is structurally equivalent to a term of the kind $\mathbf{e}[(p \mapsto d) \Rsh a_p]$. To simplify the presentation, we omit the proof concerning the conditions on the term P, but they can be easily obtained using the same arguments than in the proof of Th. 4.1.

case $a = u$**:** in this case $[\![a]\!]$ does not reduces. The proof is similar in the case such that a is a denomination $(p \mapsto d)$, a method select, a method update or a cloning;

case $a = (\nu p)b$**:** the result follows from the fact that $[\![a]\!] = (\nu p)[\![b]\!]$ and that $(\nu p)P$ reduces if and only if P does;

case $a = (\mathbf{let}\, x = b\, \mathbf{in}\, c)$**:** suppose that $[\![a]\!] \rightarrow_{(\rho)} P$. It is easy to show that there exits a term Q such that $[\![b]\!] \rightarrow_{(\rho)} Q$. Therefore, using the induction hypothesis, it follows that there exists an evaluation context $\mathbf{e}$ and a name p such that $b \equiv \mathbf{e}[(p \mapsto d) \Rsh b_p]$. Therefore we have that (modulo renaming of the bound names in b): $a \equiv \mathbf{f}[(p \mapsto d) \Rsh b_p]$, where $\mathbf{f}$ is the evaluation context defined by $\mathbf{f} =_{\text{def}} (\mathbf{let}\, x = \mathbf{e}\, \mathbf{in}\, c)$;

case $a = (b \Rsh c)$**:** suppose that $[\![a]\!] \rightarrow_{(\rho)} P$. A communication is a reduction involving both a declaration $\langle p \Leftarrow R \rangle$, and an occurrence of the name p in head position. Therefore, in the interpretation of a term a, a communication is possible only if a contains both a denomination $(p \mapsto d)$ and a term a_p in parallel. Indeed, in the interpretation of **concς**, declarations are only used in the encoding of the denomination and the **let** constructs and, in the last case, the communication can occur only after a β-reduction step (this result is along the line in the proof of Lemma 4.1). Therefore it is easy to show that there are two cases.

- there exists a term Q such that either $[\![b]\!] \rightarrow_{(\rho)} Q$ or $[\![c]\!] \rightarrow_{(\rho)} Q$. The result follows from the induction hypothesis and the fact that, if $\mathbf{e}$ is an evaluation contexts, then $(\mathbf{e} \Rsh c)$ and $(b \Rsh \mathbf{e})$ are two evaluation contexts;
- there exists two evaluation contexts, $\mathbf{e_1}$ and $\mathbf{e_2}$, that do not bound the name p and such that either $b \equiv \mathbf{e_1}[(p \mapsto d)]$ and $c \equiv \mathbf{e_2}[a_p]$, or $b \equiv \mathbf{e_1}[a_p]$ and $c \equiv \mathbf{e_2}[(p \mapsto d)]$. Therefore, using α-conversion to rename the bound name of b and c, it follows that there exists an evaluation context $\mathbf{e}$ such that $a \equiv \mathbf{e}[(p \mapsto d) \Rsh a_p]$.

□

In Lemma 4.1 and 4.2, we have drawn a parallel between β-reduction in $\boldsymbol{\pi}^\star$ and reduction inside **let**-constructs, *see* rule (gh red let result) in Table 5, and between communication and the other reduction rules of **concς**. We can prove that our interpretation of **concς** is sound using these properties,.

Theorem 4.2 (Soundness). *If* $[\![a]\!] \rightarrow P$ *in* $\boldsymbol{\pi}^\star$*, then there exists a term* a' *in* ***concς*** *such that* $a \rightarrow a'$ *and* $P \overset{*}{\rightarrow} \sim_b [\![a']\!]$*.*

Proof. Suppose that $[\![a]\!] \rightarrow P'$, therefore either it is a β-reduction: $[\![a]\!] \rightarrow_{(\beta)} P'$, or it is a communication: $[\![a]\!] \rightarrow_{(\rho)} P'$. In the first case, the result follows directly from Lemma 4.1. In the second case, we use Lemma 4.2 to prove that there exists an evaluation context $\mathbf{e}$ such that $a \equiv \mathbf{e}[(p \mapsto d) \Rsh a_p]$. The expected result follows using Th. 4.1. □

5 Typing Objects

In this section, we establish a derived type system for objects using the definition given in Table 6. The type system used in the target calculus, defined in Sect. 5.2, is a first-order type system with recursion and a special type constructor for continuations.

5.1 Simple Type System for concς

The types defined in [21] consist of the first-order objects types of Abadi and Cardelli's $\mathbf{Ob}_{1<:}$ extended with new type constants for expressions, processes and synchronization. In this type system, a clear distinction is made between *expressions*, ie. terms expected to return results, and *processes*, that intuitively represents the store of an expression. Then, the type system is used for two different goals.

1. To guarantee that terms are well-formed and in particular that a name cannot be associated to two different denomination. This situation is comparable to the use of type systems to avoid two different objects to be defined on the same reference [3];
2. to avoid "runtime errors", that are instances of the so-called "message not understood" problem.

In this section we define a simplified version of this type system that only guarantees the safety of execution (ie. goal 2). As in $\mathbf{Ob}_{1<:}$, the basic type constructor is $[\,l_i : {A_i}^{i\in[1..n]}\,]$, the type of objects with methods $(l_i)_{i\in[1..n]}$ returning results of types $(A_i)_{i\in[1..n]}$ respectively. Object types are identified up-to reordering of their components. There is also a type constant, *Proc*, used to type processes, like denominations for example.

Definition 5.1 (Types and Environments).

$$A, B ::= [\,l_i : {A_i}^{i\in[1..n]}\,] \;\;\mathbf{|}\;\; \mathit{Proc} \qquad\qquad E ::= \emptyset \;\mathbf{|}\; E, u : A$$

The type system is based on four judgments, defined by the rules given in Table 7. These judgments are respectively:

- $E \vdash \diamond$: E is a well-formed environment;
- $E \vdash A$: given E, type A is well-formed;
- $E \vdash A <: B$: given E, type A is a subtype of B;
- $E \vdash a : A$: given E, term a has type A.

Note that, as in $\mathbf{Ob}_{1<:}$, there is no "depth subtyping" between object types.

5.2 Simple Type System for $\pi^\star$

We define a first-order type system for the blue calculus that directly embeds the simple type system "à la Curry" for the λ-calculus. It is essentially the type system given in [8] and extended with record types, recursion and subtyping. Another extension lies in the definition of a special operator $\mathit{Reply}(\tau)$, used to type continuations. This operator is thoroughly described in Sect. 5.2.

We assume given a denumerable set of type variables ranged over by $\alpha, \beta, \ldots$. The syntax of type expressions, denoted $\tau, \sigma, \ldots$, is given in Definition 5.3. As in **concς**, the type system is based on four judgments:

- $\Gamma \vdash \diamond$: Γ is a well-formed environment;
- $\Gamma \vdash \tau :: \kappa$: given Γ, type τ has kind κ;
- $\Gamma \vdash \tau <: \vartheta$: given Γ, type τ is a subtype of ϑ;
- $\Gamma \vdash P : \tau$: given Γ, term P has type τ;

In the remainder of this paper, we consider that types are well formed with respect to the kinding system defined by the rules in Table 8. This is the main difference with the system of **concς**, where there is only one (implicit) kind. We consider two kinds: $\mathbb{R}$ for rows and $\mathbb{T}$ for functional types.

Let A be the object type $[\,l_i : B_i{}^{i \in [1..n]}\,]$, and let $(l_i)_{i \in [1..n]}$ be distinct method names.

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{E \vdash B \quad u \notin \mathbf{dom}(E)}{E, u : B \vdash \diamond} \qquad \frac{E \vdash \diamond}{E \vdash \mathit{Proc}} \qquad \frac{E \vdash \diamond \quad E \vdash B_i \quad B_i \neq \mathit{Proc} \quad \forall i \in [1..n]}{E \vdash [\,l_i : B_i{}^{i \in [1..n]}\,]}$$

$$\frac{E \vdash B}{E \vdash B <: B} \qquad \frac{E \vdash B_1 <: B_2 \quad E \vdash B_2 <: B_3}{E \vdash B_1 <: B_3} \qquad \frac{E \vdash [\,l_i : B_i{}^{i \in [1..n+m]}\,]}{E \vdash [\,l_i : B_i{}^{i \in [1..n+m]}\,] <: [\,l_i : B_i{}^{i \in [1..n]}\,]} \text{ (gh sub obj)}$$

$$\frac{E \vdash B}{E \vdash B <: \mathit{Proc}} \text{ (gh sub proc)} \qquad \frac{E \vdash \diamond \quad (u : B) \in E}{E \vdash u : B} \text{ (gh val } u\text{)} \qquad \frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B} \text{ (gh val sub)}$$

$$\frac{E, x_i : A \vdash b_i : B_i \quad \forall i \in [1..n] \quad (p : A) \in E}{E \vdash (p \mapsto [\,l_i = \varsigma(x_i)b_i{}^{i \in [1..n]}\,]) : \mathit{Proc}} \text{ (gh val obj)} \qquad \frac{E \vdash u : A \quad j \in [1..n]}{E \vdash u{\cdot}l_j : B_j} \text{ (gh val invk)}$$

$$\frac{E, x : A \vdash b : B_j \quad E \vdash u : A \quad j \in [1..n]}{E \vdash u{\cdot}l_j \Leftarrow \varsigma(x)b : A} \text{ (gh val updt)} \qquad \frac{E \vdash u : A}{E \vdash \mathbf{clone}(u) : A} \text{ (gh val clone)}$$

$$\frac{E \vdash a : B \quad A, B \neq \mathit{Proc} \quad E, x : B \vdash b : C}{E \vdash \mathbf{let}\, x = a \,\mathbf{in}\, b : C} \text{ (gh val let)} \qquad \frac{E \vdash a : \mathit{Proc} \quad E \vdash b : B}{E \vdash a \Rsh b : B} \qquad \frac{E, p : B \vdash a : C}{E \vdash (\nu p) a : C}$$

Table 7: Typing Rules for **concς**

Definition 5.2 (Kinds). $\kappa, \chi ::= \mathbb{R} \mid \mathbb{T}$

In this system the empty row [ ] has kind $\mathbb{R}$, while *Top* have kind $\mathbb{T}$. Intuitively, the kind system is used to constrain the type ϱ, in the extension $[\,\varrho\,,\, l : \tau\,]$, to be of kind $\mathbb{R}$. That is ϱ is a row. In particular we rule out types such as $[\,\mathit{Top}\,,\, l : \tau\,]$.

Definition 5.3 (Types and Environments). *We denote* $[\,l_1 : \tau_1, \ldots, l_n : \tau_n\,]$ *the* row*:* $[\,[\,[\,]\,, l_1 : \tau_n\,]\,, \ldots, l_n : \tau_n\,]$*, whenever the fields in* $(l_i)_{i \in [1..n]}$ *are distinct. Types and environments are generated by the following grammar:*

$$\begin{array}{rcll} \tau, \vartheta, \varrho & ::= & \alpha & \text{type variable} \\ & \mid & \mathit{Top} & \text{maximal type} \\ & \mid & (\tau \to \vartheta) & \text{function type} \\ & \mid & (\mu\alpha.\tau) & \text{recursive type} \\ & \mid & \mathit{Reply}(\tau) & \text{continuation type} \\ & \mid & [\,] & \text{empty row} \\ & \mid & [\,\varrho\,,\, l : \tau\,] & \text{row update/extension} \end{array} \qquad \mathit{\Gamma}, \mathit{\Delta} ::= \emptyset \mid \mathit{\Gamma}, a : \tau \mid \mathit{\Gamma}, \alpha :: \kappa$$

The operators given in Definition 5.3 are all borrowed from type systems for the λ-calculus, apart for *Reply*(.), that is used to type the continuation operator: **reply**(P), and linear application: $(\mathbf{set}\, x = P \;\mathbf{in}\, Q)$. The type *Top*, for example, is the maximal type with respect to the subtyping relation. Nonetheless we make a non-standard use of this type constant. Indeed *Top* is used to type terms that may not be expected to return results, like for the process type *Proc* in **concς**, and it is, for example, the type given to resources, *see* rule (type decl) in Table 10.

Subtyping The subtyping relation is defined inductively by the rules given in Table 9. The rules for the functional part of the system are classical. In particular arrow types $(\tau \to \vartheta)$,

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{\Gamma \vdash \tau :: \mathbb{T} \quad u \notin \mathbf{dom}(\Gamma)}{\Gamma, x : \tau \vdash \diamond} \qquad \frac{\Gamma \vdash \diamond \quad \alpha \notin \mathbf{fn}(\Gamma)}{\Gamma, \alpha :: \kappa \vdash \diamond}$$

$$\frac{\Gamma \vdash \tau :: \mathbb{R}}{\Gamma \vdash \tau :: \mathbb{T}} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash [\,] :: \mathbb{R}} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Top} :: \mathbb{T}} \qquad \frac{\Gamma \vdash \diamond \quad (\alpha :: \kappa) \in \Gamma}{\Gamma \vdash \alpha :: \kappa} \qquad \frac{\Gamma, \alpha :: \mathbb{T} \vdash \tau :: \mathbb{T}}{\Gamma \vdash \mu\alpha.\tau :: \mathbb{T}}$$

$$\frac{\Gamma \vdash \tau :: \mathbb{T}}{\Gamma \vdash \mathit{Reply}(\tau) :: \mathbb{T}} \qquad \frac{\Gamma \vdash \tau :: \mathbb{T} \quad \Gamma \vdash \vartheta :: \mathbb{T}}{\Gamma \vdash (\tau \to \vartheta) :: \mathbb{T}} \qquad \frac{\Gamma \vdash \varrho :: \mathbb{R} \quad \Gamma \vdash \tau :: \mathbb{T}}{\Gamma \vdash [\,\varrho\,,\, l : \tau\,] :: \mathbb{R}}$$

Table 8: Well-formed Types and Environments

are *contravariant* in the first parameter and *covariant* in the second. For commodity reason, we note $\sim$ the equivalence relation defined as $\Gamma \vdash \tau \sim \vartheta$, if and only if $\Gamma \vdash \tau <: \vartheta$ and $\Gamma \vdash \vartheta <: \tau$. We simply write this relation $\tau \sim \vartheta$ when the environment it deducible from the context. For example, using the rules for recursion folding/unfolding, it is easy to show that $\mu\alpha.\tau \sim \tau\{{}^{\mu\alpha.\tau}/_{\alpha}\}$. Rule (sub cont) states that the continuation type operator $\mathit{Reply}(\tau)$ is covariant.

$$\frac{\Gamma \vdash \tau :: \mathbb{T}}{\Gamma \vdash \tau <: \tau}\ \text{(sub refl)} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}\ \text{(sub trans)} \qquad \frac{\Gamma \vdash \tau :: \mathbb{T}}{\Gamma \vdash \tau <: \mathit{Top}}\ \text{(sub top)}$$

$$\frac{\Gamma \vdash [\,\varrho\,,\, l : \tau\,] :: \mathbb{R}}{\Gamma \vdash [\,\varrho\,,\, l : \tau\,] <: [\,]}\ \text{(sub void)} \qquad \frac{\begin{array}{c}\Gamma, \alpha :: \mathbb{T} \vdash \tau :: \mathbb{T} \quad \Gamma, \beta :: \mathbb{T} \vdash \vartheta :: \mathbb{T} \\ \alpha <: \beta \Rightarrow \Gamma \vdash \tau <: \vartheta\end{array}}{\Gamma \vdash \mu\alpha.\tau <: \mu\beta.\vartheta}\ \text{(sub rec)}$$

$$\frac{\Gamma \vdash \mu\alpha.\tau :: \mathbb{T}}{\Gamma \vdash \tau\{{}^{\mu\alpha.\tau}/_{\alpha}\} <: \mu\alpha.\tau}\ \text{(sub rec fold)} \qquad \frac{\Gamma \vdash \mu\alpha.\tau :: \mathbb{T}}{\Gamma \vdash \mu\alpha.\tau <: \tau\{{}^{\mu\alpha.\tau}/_{\alpha}\}}\ \text{(sub rec unfold)}$$

$$\frac{\Gamma \vdash \tau <: \vartheta}{\Gamma \vdash \mathit{Reply}(\tau) <: \mathit{Reply}(\vartheta)}\ \text{(sub cont)} \qquad \frac{\Gamma \vdash [\,[\,\varrho\,,\, k : \vartheta\,]\,,\, l : \tau\,] :: \mathbb{R} \qquad k \neq l}{\Gamma \vdash [\,[\,\varrho\,,\, k : \vartheta\,]\,,\, l : \tau\,] <: [\,[\,\varrho\,,\, l : \tau\,]\,,\, k : \vartheta\,]}\ \text{(sub swap)}$$

$$\frac{\Gamma \vdash \vartheta_1 <: \tau_1 \quad \Gamma \vdash \tau_2 <: \vartheta_2}{\Gamma \vdash \tau_1 \to \tau_2 <: \vartheta_1 \to \vartheta_2}\ \text{(sub arrow)} \qquad \frac{\begin{array}{c}\Gamma \vdash [\,\varrho\,,\, l : \tau\,] :: \mathbb{R} \\ \Gamma \vdash \varrho <: \varrho' \quad \Gamma \vdash \tau <: \tau'\end{array}}{\Gamma \vdash [\,\varrho\,,\, l : \tau\,] <: [\,\varrho'\,,\, l : \tau'\,]}\ \text{(sub over)}$$

Table 9: Subtyping Rules

The subtyping rules for rows are less classical, and reflect their incremental construction. In particular the rule (sub swap) allows to identify rows up-to reordering of their components, that is, we have: $[\ldots, l : \tau, k : \vartheta, \ldots] \sim [\ldots, k : \vartheta, l : \tau, \ldots]$. We can also retrieve the classical notions of with and depth subtyping.

Lemma 5.1 (Width Subtyping). $[\,l_1 : \tau_1, \ldots, l_{n+k} : \tau_{n+k}\,] \ <: \ [\,l_1 : \tau_1, \ldots, l_n : \tau_n\,]$

Proof. By induction on k, using rules (sub swap), (sub void) and (sub over). For example in the case $k = n = 1$, the judgment $[\,l_1 : \tau_1, l_2 : \tau_2\,] <: [\,l_1 : \tau_1\,]$ is obtained by the fact that $[\,[\,[\,]\,,\, l_1 : \tau_1\,]\,,\, l_2 : \tau_2\,] <: [\,[\,[\,]\,,\, l_2 : \tau_2\,]\,,\, l_1 : \tau_1\,]$ and $[\,[\,]\,,\, l_2 : \tau_2\,] <: [\,]$. □

Lemma 5.2 (Depth Subtyping). *If for all index i in $[1..n]$ we have $\tau_i <: \vartheta_i$, then $[\,l_1 : \tau_1, \ldots, l_n : \tau_n\,] \;<:\; [\,l_1 : \vartheta_1, \ldots, l_n : \vartheta_n\,]$.*

Proof. By induction on n, using rule (sub over). □

These relations are two instances of a more general property that characterize the cases such that two rows, given in normal form $[\,l_i : {\tau_i}^{i \in I}\,]$, are in the subtype relation.

Lemma 5.3 (Subtyping). *If $\Gamma \vdash [\,\varrho\,,\, l : \tau\,] <: [\,k : \vartheta\,]$ and $k \neq l$, then $\Gamma \vdash \varrho <: [\,k : \vartheta\,]$. If $\Gamma \vdash [\,\varrho\,,\, l : \tau\,] <: [\,\varrho'\,,\, l : \vartheta\,]$, then $\Gamma \vdash \tau <: \vartheta$. The relation $[\,l_i : {\tau_i}^{i \in I}\,] <: [\,l_i : {\vartheta_i}^{i \in J}\,]$ holds if and only if J is included in I and, for all $i \in J$, we have $\tau_i <: \vartheta_i$.*

Proof. The proof of these properties is along the lines in [14]. □

Typing rules The typing rules are given in Table 10. We suppose that types are defined up-to α-conversion of terms. This is needed, for example, to type the term $(\lambda x)(\lambda x)P$.

$$\frac{\Gamma \vdash \diamond \quad (u : \tau) \in \Gamma}{\Gamma \vdash u : \tau}\ \text{(type ax)} \qquad \frac{\Gamma, x : \tau \vdash P : \vartheta}{\Gamma \vdash (\lambda x)P : \tau \to \vartheta}\ \text{(type abs)}$$

$$\frac{\Gamma \vdash P : \tau \to \vartheta \quad \Gamma \vdash u : \tau}{\Gamma \vdash (Pu) : \vartheta}\ \text{(type app)} \qquad \frac{\Gamma \vdash P : [\,l : \tau\,]}{\Gamma \vdash (P \cdot l) : \tau}\ \text{(type sel)}$$

$$\frac{\Gamma \vdash P : \varrho \quad \Gamma \vdash Q : \tau}{\Gamma \vdash [\,P\,,\, l = Q\,] : [\,\varrho\,,\, l : \tau\,]}\ \text{(type over)} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash [\,] : [\,]}\ \text{(type void)}$$

$$\frac{\Gamma \vdash P : \tau \quad \Gamma, \Gamma' \vdash \diamond}{\Gamma, \Gamma' \vdash P : \tau}\ \text{(type weak)} \qquad \frac{\Gamma \vdash P : \tau \quad \Gamma \vdash \tau <: \vartheta}{\Gamma \vdash P : \vartheta}\ \text{(type sub)}$$

$$\frac{\Gamma, u : \tau \vdash P : \vartheta}{\Gamma \vdash (\nu u)P : \vartheta}\ \text{(type new)} \qquad \frac{\Gamma \vdash P : \mathit{Top} \quad \Gamma \vdash Q : \tau}{\Gamma \vdash (P \mid Q) : \tau}\ \text{(type par)}$$

$$\frac{\Gamma \vdash P : \tau \quad (u : \tau) \in \Gamma}{\Gamma \vdash \langle u {\Leftarrow} P \rangle : \mathit{Top}}\ \text{(type decl)} \qquad \frac{\Gamma \vdash P : \tau \quad (u : \tau) \in \Gamma}{\Gamma \vdash \langle u {=} P \rangle : \mathit{Top}}\ \text{(type mdecl)}$$

$$\frac{\Gamma \vdash P : \tau}{\Gamma \vdash \mathbf{reply}(P) : \mathit{Reply}(\tau)}\ \text{(type reply)} \qquad \frac{\Gamma \vdash P : \mathit{Reply}(\tau) \quad \Gamma \vdash u : \tau \to \vartheta}{\Gamma \vdash (Pu) : \vartheta}\ \text{(type cont)}$$

Table 10: Typing Rules

The typing rules for the functional part of the calculus are those of the simply typed λ-calculus extended with records and subtyping. We may explain the typing rules for the π-calculus operators as follows. A parallel composition has type τ if, in the term $(P \mid Q)$, the "main thread" of computation (ie. Q) has type τ, and if the term P, that represents the environment of Q, has type *Top*. Note that, since Q has the type *Top*, its only possible interactions with the ouside world are by communication. In particular it is impossible to apply Q to a name. This motivates the choice of the maximal type *Top* for typing processes.

The typing rules for declarations, (type decl) and (type mdecl), deserves more comment. The sequent $\Gamma, u : \tau, \Gamma' \vdash P : \vartheta$ denotes that the name u is used in P with the type τ. Suppose that u also appears in subject position of a declaration $\langle u {\Leftarrow} Q \rangle$, for example the term P is equal to $(\langle u {\Leftarrow} Q \rangle \mid R)$. Since Q may be substituted to the occurrence of u in P (*see* rule (red decl) in Table 2), the term Q must have the type τ. This is the same argument as in the typing of the "let" construct in ML. The type of a resource is the type of the name used to access it.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash P : \tau} \qquad (u : \tau) \in \Gamma}{\Gamma \vdash \langle u{=}P\rangle : \mathit{Top}}\;\text{(type mdecl)} \qquad\qquad \dfrac{\vdots}{\Gamma \vdash Q : \vartheta}}{\Gamma \vdash (\langle u{=}P\rangle \mid Q) : \vartheta}\;\text{(type par)} \tag{22}$$

The same derivation applies for the term $(\langle u{\Leftarrow}Q\rangle \mid P)$. Using this derivation, it is easy to derive typing rules for definition and higher-order application.

$$\boxed{\dfrac{\Gamma, u : \tau \vdash P : \tau \qquad \Gamma, u : \tau \vdash Q : \sigma}{\Gamma \vdash \mathbf{def}\, u = P \,\mathbf{in}\, Q : \sigma}\;\text{(type def)} \qquad \dfrac{\Gamma \vdash P : \tau \to \vartheta \qquad \Gamma \vdash Q : \tau}{\Gamma \vdash (P\, Q) : \vartheta}\;\text{(type app)}}$$

Typing Continuations We recall that $\mathbf{reply}(a)$ stands for the term $(\lambda r)(ra)$. In this paragraph, we explain the rule for typing and subtyping the operator $\mathit{Reply}(.)$. Let α be a fresh type variable. Using rules (type weak), (type app) and (type abs), it is easy to show that the following judgment is valid:

$$\dfrac{\Gamma \vdash P : \tau}{\Gamma, \alpha :: \mathbb{T} \vdash (\lambda r)(r\, P) : (\tau \to \alpha) \to \alpha}$$

Note that the type $(\tau \to \alpha) \to \alpha$ is peculiar to typed "continuation passing style" transformations in the λ-calculus [25, 8]. In $(\tau \to \alpha) \to \alpha$, the variable α is implicitly quantified, in the sense that $\mathbf{reply}(P)$ has type $\forall\alpha.((\tau \to \alpha) \to \alpha)$ in the Hindley-Milner type system [15]. To avoid the introduction of quantification in our type system, we use a new type operator (namely $\mathit{Reply}(\tau)$) to type the term $\mathbf{reply}(P)$. This mechanism is similar to the use of the $\mathbf{let}\, x = N \,\mathbf{in}\, M$ construct in ML, that is (operationally) a shorthand for the term $(\lambda x.M\, N)$, but that is used to introduce polymorphic types. Using the interpretation of $\mathit{Reply}(\tau)$ as the type $(\tau \to \alpha) \to \alpha$, it is easy to (informally) validate the rules (type cont) and (sub cont). Indeed it is easy to show that $\Gamma, \alpha :: \mathbb{T} \vdash P : (\tau \to \alpha) \to \alpha$ implies that, for all well-formed type ϑ, we have $\Gamma \vdash P : (\tau \to \vartheta) \to \vartheta$. Therefore:

$$\dfrac{\begin{array}{c}\Gamma, \alpha :: \mathbb{T} \vdash P : (\tau \to \alpha) \to \alpha \\ \Gamma \vdash u : (\tau \to \vartheta)\end{array}}{\Gamma \vdash (Pu) : \vartheta} \qquad \text{and} \qquad \dfrac{\Gamma \vdash \tau <: \vartheta}{\Gamma, \alpha :: \mathbb{T} \vdash (\tau \to \alpha) \to \alpha <: (\vartheta \to \alpha) \to \alpha}$$

Using rule (type cont) and the derivation given in (22), we can derive a very simple typing rule for linear application $(\mathbf{set}\, x = P \;\mathbf{in}\, Q)$, that is for the term $(\nu u)(\langle u{\Leftarrow}(\lambda x)Q\rangle \mid P\, u)$:

$$\boxed{\dfrac{\Gamma \vdash P : \mathit{Reply}(\tau) \qquad \Gamma, x : \tau \vdash Q : \vartheta}{\Gamma \vdash \mathbf{set}\, x = P \;\mathbf{in}\, Q : \vartheta}\;\text{(type setin)}}$$

In Sect. 5.3, the operator $\mathit{Reply}(\tau)$ will play a central role in the interpretation of **concς** types in the blue calculus.

It is easy to show that the typing rules given in Table 10 define an algorithm for deciding wether a $\boldsymbol{\pi}^\star$ term is well-typed. This algorithm only use resolution of subtyping constraints that are comparable to those used for solving typechecking problem in the λ-calculus with subtyping [35]. We can also show that the blue calculus is type safe. Indeed, using the results given in [14], where the author proves the subject property for an implicit type system with subtyping and higher-order quantification, we can prove that the typing rules respect structural congruence and reduction. That is:

Theorem 5.1 (Subject Reduction). *If $\Gamma \vdash P : \tau$ and $P \equiv Q$, then $\Gamma \vdash Q : \tau$. If $\Gamma \vdash P : \tau$ and $P \to P'$, then $\Gamma \vdash P' : \tau$.*

5.3 Interpretation of concς Types

We prove a type correspondence between **concς** and the blue calculus. To simplify our presentation, we first define a special notation for the type of a denomination. Note that this type is very similar to the one obtained in the encoding of Abadi and Cardelli's functional object calculus given by Viswanathan [40] and Sangiorgi [36].

Definition 5.4 (Object Type). *We denote* $\mathrm{Obj}(\alpha.[\,l_i : \vartheta_i^{i\in[1..n]}\,])$ *the recursive type defined below. In this type, the variable α is called the* self-type. *We use the notation* $\mathrm{Obj}([\,l_i : \vartheta_i^{i\in[1..n]}\,])$ *whenever the self-type is not in the* $(\vartheta_i)_{i\in[1..n]}$.

$$\mathrm{Obj}(\alpha.[\,l_i : \vartheta_i^{i\in[1..n]}\,]) =_{\mathrm{def}} \mu\alpha. \left[\begin{array}{ll} \ldots & {}^{i\in[1..n]} \\ \mathit{get}_{l_i} = \vartheta_i, & \\ \mathit{put}_{l_i} = (\alpha \to \vartheta_i) \to \mathit{Reply}(\alpha), & \\ \ldots & \\ \mathit{clone} = \mathit{Reply}(\alpha) & \end{array}\right]$$

Let $\tilde{x}$ be the tuple of variables $(x_1, \ldots, x_n)$ and ϱ be the type $[\,l_i : \vartheta_i^{i\in[1..n]}\,]$. We show that the type of the record $\mathsf{R}(p, s, \tilde{x}, c)$ (see (18) in Sect. 4) is $\mathrm{Obj}(\alpha.\varrho)$, under the hypothesis that the names x_i, for $i \in [1..n]$, have type $(\alpha{\to}\vartheta_i)\{\mathrm{Obj}(\alpha.\varrho)/_{\alpha}\}$. Let Γ be the environment defined by:

$$\begin{aligned} \Gamma =_{\mathrm{def}}\ & \Delta, p : \tau, c : (\tau \to \vartheta_1\{\tau/_{\alpha}\}) \to \cdots \to (\tau \to \vartheta_n\{\tau/_{\alpha}\}) \to \mathit{Reply}(\tau), \\ & s : (\tau \to \vartheta_1\{\tau/_{\alpha}\}) \to \cdots \to (\tau \to \vartheta_n\{\tau/_{\alpha}\}) \to \mathit{Top}, \\ & x_1 : (\tau \to \vartheta_1\{\tau/_{\alpha}\}), \ldots, x_n : (\tau \to \vartheta_n\{\tau/_{\alpha}\}) \end{aligned}$$

In the environment Γ, the term $(s\tilde{x})$ has the type of processes, that is: $\Gamma \vdash s\tilde{x} : \mathit{Top}$, whereas $(c\tilde{x})$ has type $\mathit{Reply}(\tau)$, and $(x_i e)$ has type $\vartheta_i\{\tau/_{\alpha}\}$. Therefore we have:

$$\Gamma \vdash \mathsf{R}(p, s, \tilde{x}, c) : \left[\begin{array}{ll} \ldots & {}^{i\in[1..n]} \\ \mathit{get}_{l_i} = \vartheta_i\{\tau/_{\alpha}\}, & \\ \mathit{put}_{l_i} = (\tau \to \vartheta_i\{\tau/_{\alpha}\}) \to \mathit{Reply}(\tau), & \\ \ldots & \\ \mathit{clone} = \mathit{Reply}(\tau) & \end{array}\right]$$

This derivation is also valid if we replace τ by the type $\mathrm{Obj}(\alpha.\varrho)$, therefore we have:

$$\Gamma \vdash \mathsf{R}(p, s, \tilde{x}, c) : \left[\begin{array}{ll} \ldots & {}^{i\in[1..n]} \\ \mathit{get}_{l_i} = \vartheta_i, & \\ \mathit{put}_{l_i} = (\alpha \to \vartheta_i) \to \mathit{Reply}(\alpha), & \\ \ldots & \\ \mathit{clone} = \mathit{Reply}(\alpha) & \end{array}\right] \{\mathrm{Obj}(\alpha.\varrho)/_{\alpha}\} \qquad (23)$$

That is, by unfolding the recursive type definition, we have shown that $\mathsf{R}(p, s, \tilde{x}, c)$ has the type $\mathrm{Obj}(\alpha.\varrho)$. It is easy to understand why we need recursion to type objects. Indeed, in the definition of $\langle p \mapsto L \rangle$ we use the declaration $\langle p \Leftarrow \mathsf{R}(p, s, \tilde{x}, c) \rangle$ (*see* (20)). Therefore, a necessary condition for an object to be well-typed, is that $\mathrm{Obj}(\alpha.\varrho)$ is also the type of p.

In the remainder of this section we define the typing rules for the derived object operator of $\boldsymbol{\pi}^{\star}$. These typing rules are collected in Table 11. To simplify the proof, we denote Γ the following environment:

$$\begin{aligned} \Gamma =_{\mathrm{def}} \;& \Delta, c : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \mathit{Reply}(\alpha))\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}, \\ & s : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \mathit{Top})\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}, \\ & x_1 : (\alpha \to \vartheta_1)\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}, \ldots, x_n : (\alpha \to \vartheta_n)\{\mathrm{Obj}(\alpha.\varrho)/\alpha\} \end{aligned}$$

We also suppose that $(p : \mathrm{Obj}(\alpha.\varrho)) \in \Delta$, and that $\Gamma \vdash \diamond$.

Typing denominations. Using the judgment given in (23) and rule (type decl), it is easy to show that $\Gamma \vdash \langle p \Leftarrow \mathsf{R}(p, s, \tilde{x}, c) \rangle : \mathit{Top}$. Therefore, using rule (type abs), we prove that:

$$\Gamma_{|\tilde{x}} \vdash (\lambda\tilde{x})\langle p \Leftarrow \mathsf{R}(p, s, \tilde{x}, c) \rangle : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \mathit{Top})\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}$$

That is the type of s. Therefore we have:

$$\begin{array}{l} \Gamma_{|s,\tilde{x}}, u_1 : (\alpha \to \vartheta_1)\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}, \\ \ldots, u_n : (\alpha \to \vartheta_n)\{\mathrm{Obj}(\alpha.\varrho)/\alpha\} \end{array} \vdash \left(\begin{array}{l} \mathbf{def}\, s = (\lambda\tilde{x})\langle p \Leftarrow \mathsf{R}(p, s, \tilde{x}, c) \rangle \\ \mathbf{in}\, \langle p \Leftarrow \mathsf{R}(p, s, \tilde{u}, c) \rangle \end{array} \right) : \mathit{Top} \tag{24}$$

And thus the type of $\mathbf{\mathsf{Fobj}}(p, \tilde{x}, c)$ is Top. This last judgment implies that the term $(\lambda\tilde{x})(\nu q)(\mathbf{\mathsf{Fobj}}(q, \tilde{x}, c) \mid \mathbf{reply}(q))$ has the type of the name c in Γ. Indeed we have:

$$\begin{array}{c} \Gamma_{|s,\tilde{x}}, x_1 : (\alpha \to \vartheta_1)\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}, \ldots, q : \mathrm{Obj}(\alpha.\varrho) \vdash \mathbf{\mathsf{Fobj}}(q, \tilde{x}, c) : \mathit{Top} \\ \hline \Gamma_{|s,\tilde{x}}, x_1 : (\alpha \to \vartheta_1)\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}, \ldots, q : \mathrm{Obj}(\alpha.\varrho) \vdash (\mathbf{\mathsf{Fobj}}(q, \tilde{x}, c) \mid \mathbf{reply}(q)) : \mathit{Reply}(\mathrm{Obj}(\alpha.\varrho)) \\ \hline \Gamma_{|s,\tilde{x}} \vdash (\lambda\tilde{x})(\nu q)(\mathbf{\mathsf{Fobj}}(q, \tilde{x}, c) \mid \mathbf{reply}(q)) : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \mathit{Reply}(\alpha))\{\mathrm{Obj}(\alpha.\varrho)/\alpha\} \end{array}$$

In particular, if $\Gamma_{|s,\tilde{x}} \vdash P : \tau$, it follows from rule (type def) that:

$$\Gamma_{|s,\tilde{x}} \vdash \mathbf{def}\, c = (\lambda\tilde{x})(\nu q)(\mathbf{\mathsf{Fobj}}(q, \tilde{x}, c) \mid \mathbf{reply}(q))\, \mathbf{in}\, P : \tau$$

Therefore, if for every index i in $[1..n]$ we have that $\Delta \vdash (\lambda x_i)P_i : (\alpha \to \vartheta_i)\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}$, we can conclude that:

$$\Delta \vdash \left(\begin{array}{l} \mathbf{def}\, c = (\lambda\tilde{x})(\nu q)(\mathbf{\mathsf{Fobj}}(q, \tilde{x}, c) \mid \mathbf{reply}(q)) \\ \mathbf{in}\;\mathbf{def}\, u_1 = (\lambda x_1)P_1, \ldots, u_n = (\lambda x_n)P_n \\ \quad \mathbf{in}\, \mathbf{\mathsf{Fobj}}(p, \tilde{u}, c) \end{array} \right) : \mathit{Top}$$

This last judgment validates rule (type obj) of Table 11.

Typing method update. We recall that $(P \cdot l_j \Leftarrow (\lambda x)Q)$ is a shorthand for the term $(P \cdot \mathit{put}_{l_j}\, (\lambda x)Q)$. Suppose that j is an index in the interval $[1..n]$ and that:

$$\Gamma \vdash P : \mathrm{Obj}(\alpha.\varrho) \qquad \text{and} \qquad \Gamma, x : \mathrm{Obj}(\alpha.\varrho) \vdash Q : \vartheta_j\{\mathrm{Obj}(\alpha.\varrho)/\alpha\}$$

Let A denotes the object type $\mathrm{Obj}(\alpha.[\,l_i : \vartheta_i^{i\in[1..n]}\,])$

$$\frac{\mathit{\Gamma}, x_i : \mathrm{A} \vdash P_i : \vartheta_i\{{}^{\mathrm{A}}/_{\alpha}\} \qquad \forall i \in [1..n] \qquad (p : \mathrm{A}) \in \mathit{\Gamma}}{\mathit{\Gamma} \vdash \langle p \mapsto \{\, l_i = (\lambda x_i) {P_i}^{i\in[1..n]} \,\}\rangle : \mathit{Top}}\ \text{(type obj)}$$

$$\frac{\mathit{\Gamma}, x : \mathrm{A} \vdash Q : \vartheta_j\{{}^{\mathrm{A}}/_{\alpha}\} \qquad \mathit{\Gamma} \vdash P : \mathrm{A} \qquad j \in [1..n]}{\mathit{\Gamma} \vdash (P \cdot l_j \Leftarrow (\lambda x) Q) : \mathit{Reply}(\mathrm{A})}\ \text{(type updt)}$$

$$\frac{\mathit{\Gamma} \vdash P : \mathrm{A}}{\mathit{\Gamma} \vdash \mathbf{clone}(P) : \mathit{Reply}(\mathrm{A})}\ \text{(type clone)} \qquad \frac{\mathit{\Gamma} \vdash P : \mathrm{A} \qquad j \in [1..n]}{\mathit{\Gamma} \vdash P \Leftarrow l_j : \vartheta_j\{{}^{\mathrm{A}}/_{\alpha}\}}\ \text{(type invk)}$$

Table 11: Typing Rules for Objects

Using the type of Q and rule (type abs), it follows that $\mathit{\Gamma} \vdash (\lambda x) Q : (\alpha \to \vartheta_j)\{{}^{\mathrm{Obj}(\alpha.\varrho)}/_{\alpha}\}$, and the type of P implies that $\mathit{\Gamma} \vdash P \cdot \mathit{put}_{l_j} : ((\alpha \to \vartheta_j) \to \mathit{Reply}(\alpha))\{{}^{\mathrm{Obj}(\alpha.\varrho)}/_{\alpha}\}$. Therefore we have:

$$\mathit{\Gamma} \vdash P \cdot \mathit{put}_{l_j}\ (\lambda x) Q : \mathit{Reply}(\mathrm{Obj}(\alpha.\varrho))$$

This judgment validates rule (type updt). Note that it is impossible to extend the object P with a new method. Indeed, the set of fields put_{l_j} is fixed. Note also that the type of the new method is exactly the one of the updated method, that is we cannot refine the type of a method. Indeed, the type of the method l_j (that is ϑ in this example) appears in contravariant position in field put_{l_j}, and in covariant position in field get_{l_j}. This mechanism will be exemplified in Sect. 5.4, where we study subtyping between object types.

Typing method invocation. We recall that $(P \Leftarrow l_j)$ is a shorthand for $(P \cdot \mathit{get}_{l_j})$. Suppose that $\mathit{\Gamma} \vdash P : \mathrm{Obj}(\alpha.\varrho)$, and that $j \in [1..n]$. It is clear that the recursive type $\mathrm{Obj}(\alpha.\varrho)$ is equivalent to a type of the kind $[\,\ldots,\ \mathit{get}_{l_j} = \vartheta_j\{{}^{\mathrm{Obj}(\alpha.\varrho)}/_{\alpha}\}\,]$. Therefore, using rule (type sel), it follows that $\mathit{\Gamma} \vdash (P \cdot \mathit{get}_{l_j}) : \vartheta_j\{{}^{\mathrm{Obj}(\alpha.\varrho)}/_{\alpha}\}$.

Typing cloning. Suppose that $\mathit{\Gamma} \vdash P : \mathrm{Obj}(\alpha.\varrho)$. Using recursion unfolding and rule (type sub), it is easy to show that $\mathit{\Gamma} \vdash P : [\,\ldots,\ \mathit{clone} = \mathit{Reply}(\mathrm{Obj}(\alpha.\varrho))\,]$, and therefore $\mathit{\Gamma} \vdash \mathbf{clone}(P) : \mathit{Reply}(\mathrm{Obj}(\alpha.\varrho))$.

With the notations introduced in this section, we can give a very direct and simple interpretation of **concς** types.

$$[\![\,[\,l_i : {A_i}^{i\in[1..n]}\,]\,]\!] = \mathrm{Obj}([\,l_i : \mathit{Reply}([\![A_i]\!])^{i\in[1..n]}\,]) \qquad [\![\mathit{Proc}]\!] = \mathit{Top} \qquad [\![\emptyset]\!] = \emptyset \qquad [\![E, u : A]\!] = [\![E]\!], u : [\![A]\!]$$

Table 12: Interpretation of **concς** Types and Environments

5.4 Subtyping Between Objects

As noted in the previous section, it is clear that an object type is not covariant. That is $\varrho <: \sigma$ does not imply $\mathrm{Obj}(\varrho) <: \mathrm{Obj}(\sigma)$. Indeed, the type of the method l_i appears covariantly in the type of the field get_{l_i} and contravariantly in the type of put_{l_i}. Nonetheless we prove a weaker notion of substitutability between object types, also known as matching, that informally amounts to prove that width-subtyping between object interfaces is sound. Let ϱ_1 and ϱ_2 be the two types $[\,l_i : \vartheta_i{}^{i\in[1..n]}\,]$ and $[\,l_i : \vartheta_i{}^{i\in[1..n+m]}\,]$. We show that a term of type $\mathrm{Obj}(\varrho_2)$ can be safely substituted to a term of type $\mathrm{Obj}(\varrho_1)$.

Theorem 5.2 (Substitutability). $\mathrm{Obj}([\,l_i : \vartheta_i{}^{i\in[1..n+m]}\,]) <: \mathrm{Obj}([\,l_i : \vartheta_i{}^{i\in[1..n]}\,])$

Proof. This proof use lemma 5.1 and the subtyping rule for recursive types (sub rec), *see* Table 9. We prove the property in the case $m = 1$, the other cases are similar.

Let α_1 and α_2 be two type variables. Under the hypothesis that $\alpha_1 <: \alpha_2$, it is easy to show that, for all type ϑ that does not contain α_1 or α_2, we have: $((\alpha_1 \to \vartheta) \to \mathit{Reply}(\alpha_1)) <: ((\alpha_2 \to \vartheta) \to \mathit{Reply}(\alpha_2))$. Therefore, it follows from Lemma 5.1 that:

$$\begin{bmatrix} get_{l_1} = \vartheta_1, \\ put_{l_1} = (\alpha_1 \to \vartheta_1) \to \mathit{Reply}(\alpha_1) \\ \dots \\ get_{l_{n+1}} = \vartheta_{n+1}, \\ put_{l_{n+1}} = (\alpha_1 \to \vartheta_{n+1}) \to \mathit{Reply}(\alpha_1) \\ clone = \mathit{Reply}(\alpha_1) \end{bmatrix} <: \begin{bmatrix} get_{l_1} = \vartheta_1, \\ put_{l_1} = (\alpha_2 \to \vartheta_1) \to \mathit{Reply}(\alpha_2) \\ \dots \\ get_{l_n} = \vartheta_n, \\ put_{l_n} = (\alpha_2 \to \vartheta_n) \to \mathit{Reply}(\alpha_2) \\ clone = \mathit{Reply}(\alpha_2) \end{bmatrix}$$

Finally we use rule (sub rec) to prove that $\mathrm{Obj}([\,l_i : \vartheta_i{}^{i\in[1..n+1]}\,]) <: \mathrm{Obj}([\,l_i : \vartheta_i{}^{i\in[1..n]}\,])$. □

Note that the derived rule given in Th. 5.2 is similar to the rule (gh sub obj) of **concς** given in Table 7.

Theorem 5.3.

- *If $E \vdash \diamond$, then $[\![E]\!] \vdash \diamond$. If $E \vdash A$, then $[\![E]\!] \vdash [\![A]\!] :: \mathbb{T}$;*
- *the interpretation preserves subtyping judgments: if $E \vdash A <: B$, then $[\![E]\!] \vdash [\![A]\!] <: [\![B]\!]$;*
- *the interpretation preserves typing judgments: if $E \vdash a : \mathit{Proc}$, then $[\![E]\!] \vdash [\![a]\!] : \mathit{Top}$. If $E \vdash a : A$ and $A \neq \mathit{Proc}$, then $[\![E]\!] \vdash [\![a]\!] : \mathit{Reply}([\![A]\!])$.*

Proof. Each of these facts can be proved by an induction on the appropriate judgment. For example the property that the interpretation preserves subtyping judgments is proved by induction of the inference of $E \vdash A <: B$. The only difficult case is for trule (gh sub obj) and is proved using Th. 5.2. The proof that the interpretation preserves typing judgments is by induction of the inference of $E \vdash a : A$. We make a case analysis on the last rule used.

(gh val u) by hypothesis we have $(u : B) \in E$. Suppose that $B = \mathit{Proc}$, therefore we have $(u : \mathit{Top}) \in [\![E]\!]$ and thus $[\![E]\!] \vdash \mathbf{reply}(u) : \mathit{Reply}(\mathit{Top})$. The result follows using rule (type sub) and the fact that $\mathit{Reply}(\mathit{Top}) <: \mathit{Top}$. In the case such that $B \neq \mathit{Proc}$, the result follows from rule (type reply). Note that if u has type Proc, then it cannot be used has the result of a **let** expression. This is a consequence of the side conditions in rule (gh val let). Therefore a name of type Proc will never participate to a reduction;

(gh val sub) the result follows using the induction hypothesis and the fact that the interpretation preserves subtyping judgments;

(gh val let) we have that $E \vdash a : B$ and $E, x : B \vdash b : C$ implies $E \vdash \mathbf{let}\, x = a \,\mathbf{in}\, b : C$, with the side condition that A and B are different from Proc. Therefore, using the induction hypothesis, it follows that $[\![E]\!] \vdash [\![a]\!] : \mathit{Reply}([\![B]\!])$ and $[\![E]\!], x : [\![B]\!] \vdash [\![b]\!] : \mathit{Reply}([\![C]\!])$. The result follows using rule (type setin). The proof is similar for the parallel composition and restriction operators;

(gh val obj) let A be the object type $[\,l_i : B_i{}^{i\in[1..n]}\,]$. By hypothesis we have that $E, x_i : A \vdash b_i : B_i$ $(\forall i \in [1..n])$ with $B_i \neq \mathit{Proc}$, and that $(p : A) \in E$, implies $E \vdash (p \mapsto [\,l_i = \varsigma(x_i)b_i{}^{i\in[1..n]}\,]) : \mathit{Proc}$. Therefore, using the induction hypothesis, we prove that that, for all indice i in $[1..n]$, we have: $[\![E]\!], x_i : [\![A]\!] \vdash \mathit{Reply}([\![B_i]\!])$. The result follows using the derived rule (type obj) given in Table 11. The proof is similar for rules (gh val invk), (gh val updt) and (gh val clone).

□

In fact we have a stronger result than Theorem 5.3. Indeed the type system for **conc**$\boldsymbol{\varsigma}$ defined in [21] does not have recursive types, nor self-types, while the derived type system given in Table 11 can capture such notions. Indeed, we can give a typed interpretation of a new version of **conc**$\boldsymbol{\varsigma}$ such that the type expressions are enriched with recursive types $(\mu X.A)$ and recursive object types $(\mathit{Obj}(X)[\,l_i : A_i{}^{i\in[1..n]}\,])$, that we can also called "self types" [1]. This new type system is defined as follow. We enrich the type system with type variables $X, Y \ldots$ and we add type variables in environments:

$$A, B ::= X \;\mid\; \mathit{Obj}(X)[\,l_i : A_i{}^{i\in[1..n]}\,] \;\mid\; \mu X.A \;\mid\; \mathit{Proc} \qquad\qquad E ::= \emptyset \;\mid\; E, u : A \;\mid\; E, X$$

The type system defined in Table 13 is exactly the one that can be derived using the rules in Table 11 and Theorem 5.2, where the interpretations of $\mu X.A$ and $\mathit{Obj}(X)[\,l_i : B_i{}^{i\in[1..n]}\,]$ are defined in the obvious way, that is, $[\![X]\!] = X$, and $[\![\mu X.A]\!] = \mu X.[\![A]\!]$, and $[\![\mathit{Obj}(X)[\,l_i : B_i{}^{i\in[1..n]}\,]]\!] = \mathrm{Obj}(X.[\,l_i : \mathit{Reply}([\![B_i]\!])^{i\in[1..n]}\,])$.

Nonetheless our encoding fell through to capture the usual subtyping rules between self-types, that is, it is generally not possible to prove that $\mathrm{Obj}(\alpha.[\,l_i : \vartheta_i{}^{i\in[1..n+m]}\,]) <: \mathrm{Obj}(\alpha.[\,l_i : \vartheta_i{}^{i\in[1..n]}\,])$, even if the type variable α appears covariantly in $\vartheta_1, \ldots, \vartheta_n$. To fix the failure of our encoding to preserve subtyping[1], we conjecture that we should refine our type system with existential types, as it is done in [1, 11].

There is another modification to **conc**$\boldsymbol{\varsigma}$ inspired by our interpretation. It consists in separating the role of process type from the role of maximal type, that is to consider two distinct type constants, *Top* and *Proc*, such that *Top* is the maximal type and that *Proc* is the type given to denominations. Indeed these two roles are collapsed in **conc**$\boldsymbol{\varsigma}$, as well as in the variant of $\boldsymbol{\pi}^\star$ defined in this paper.

6 Two Applications of our Interpretation

We have shown that reduction and type judgments of **conc**$\boldsymbol{\varsigma}$ can be derived in a rather simple and natural way in the blue calculus. Indeed denominations $\langle u \mapsto P \rangle$, appears as a particular kind of declarations, namely those that are linearly created each time they are accessed. This behavior has to be compared with the declaration $\langle u {\Leftarrow} P \rangle$, that is available only once, and with the replicated and "immutable" declaration $\langle u {=} P \rangle$. In Sect 6.1, we use our interpretation to prove some equational laws between objects using barbed congruence. In Sect. 6.2 we study how our interpretation can be extended to code the synchronization primitives described in [21].

[1] Note that **conc**$\boldsymbol{\varsigma}$ does not have self-types. Hence this failure does not affect the result given in Th. 5.3.

Let A be the object type $Obj(X)[\,l_i : B_i{}^{i\in[1..n]}\,]$, and let $(l_i)_{i\in[1..n]}$ be distinct method names.

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{E \vdash B \quad u \notin \mathbf{dom}(E)}{E, u : B \vdash \diamond} \qquad \frac{E \vdash \diamond \quad X \notin \mathbf{dom}(E)}{E, X \vdash \diamond} \qquad \frac{E \vdash \diamond}{E \vdash Proc}$$

$$\frac{E \vdash \diamond \quad X \in \mathbf{dom}(E)}{E \vdash X} \qquad \frac{E, X \vdash \diamond \quad E, X \vdash B_i \quad \forall i \in [1..n]}{E \vdash Obj(X)[\,l_i : B_i{}^{i\in[1..n]}\,]}$$

$$\frac{E \vdash B}{E \vdash B <: B} \qquad \frac{E \vdash B}{E \vdash B <: Proc} \qquad \frac{E \vdash B_1 <: B_2 \quad E \vdash B_2 <: B_3}{E \vdash B_1 <: B_3}$$

$$\frac{\forall i \in [1..n] \quad X \notin \mathbf{fn}(B_i) \quad E \vdash Obj(X)[\,l_i : B_i{}^{i\in[1..n+m]}\,]}{E \vdash Obj(X)[\,l_i : B_i{}^{i\in[1..n+m]}\,] <: Obj(X)[\,l_i : B_i{}^{i\in[1..n]}\,]}$$

$$\frac{E, X \vdash A \quad E, Y \vdash B \quad X <: Y \;\Rightarrow\; E \vdash A <: B}{E \vdash \mu X.A <: \mu Y.B}$$

$$\frac{E \vdash \diamond \quad (u : B) \in E}{E \vdash u : B} \qquad \frac{E \vdash a : B \quad E \vdash B <: C}{E \vdash a : C}$$

$$\frac{E, x_i : A \vdash b_i : B_i\{{}^{A}\!/\!{}_{X}\} \quad \forall i \in [1..n] \quad (p : A) \in E}{E \vdash (p \mapsto [\,l_i = \varsigma(x_i)b_i{}^{i\in[1..n]}\,]) : Proc} \qquad \frac{E \vdash u : A \quad j \in [1..n]}{E \vdash u{\cdot}l_j : B_j\{{}^{A}\!/\!{}_{X}\}}$$

$$\frac{E, x : A \vdash b : B_j\{{}^{A}\!/\!{}_{X}\} \quad E \vdash u : A \quad j \in [1..n]}{E \vdash u{\cdot}l_j \Leftarrow \varsigma(x)b : A} \qquad \frac{E \vdash u : A}{E \vdash \mathbf{clone}(u) : A}$$

$$\frac{E \vdash a : B \quad B, C \neq Proc \quad E, x : B \vdash b : C}{E \vdash \mathbf{let}\, x = a\, \mathbf{in}\, b : C} \qquad \frac{E \vdash a : Proc \quad E \vdash b : B}{E \vdash a \upharpoonright b : B} \qquad \frac{E, p : B \vdash a : C}{E \vdash (\nu p)a : C}$$

Table 13: Typing Rules for **conc$\boldsymbol{\varsigma}$** with recursive types and self-types

6.1 Some Equivalences on objects

Our interpretation can be used to prove that two terms are equivalent by showing that their translations are equivalent. The first law that we prove states that, under some condition, using the clone of an object is equivalent to directly use it. That is, if d is a denotation such that $p \notin \mathbf{fn}(d)$, then we have that:

$$\llbracket (\nu p)((p \mapsto d) \Rsh \mathbf{clone}(p)) \rrbracket \approx_b \llbracket (\nu p)((p \mapsto d) \Rsh p) \rrbracket \tag{25}$$

More particularly we prove that these two terms are in the expansion relation. In the proof of (25), we use the notation of Sect. 4. In particular, using the notations in (20) and (21), it follows that:

$$\begin{array}{rcll}
\llbracket (\nu p)((p \mapsto d) \Rsh \mathbf{clone}(p)) \rrbracket & = & (\nu p)(\mathbf{E_p}[\langle p \Leftarrow \mathsf{R}(p, s, \tilde{u}, c) \rangle] \mid p \cdot \mathit{clone}) & \\
& \equiv & (\nu p)\mathbf{E_p}[\langle p \Leftarrow \mathsf{R}(p, s, \tilde{u}, c) \rangle \mid p \cdot \mathit{clone}] & \\
& \gtrsim_d & (\nu p)\mathbf{E_p}[\mathsf{R}(p, s, \tilde{u}, c) \cdot \mathit{clone}] & (26) \\
& \gtrsim_d & (\nu p)\mathbf{E_p}[s\tilde{u} \mid c\tilde{u}] & (27) \\
& \gtrsim_d & (\nu p)\mathbf{E_p}[s\tilde{u} \mid ((\lambda \tilde{x})(\nu q)(\mathsf{Fobj}(q, \tilde{x}, c) \mid \mathbf{reply}(q)))\tilde{u}] & (28) \\
& \gtrsim_d & (\nu p)\mathbf{E_p}[s\tilde{u} \mid (\nu q)(\mathsf{Fobj}(q, \tilde{u}, c) \mid \mathbf{reply}(q))] & (29) \\
& \gtrsim_d & (\nu p)(\llbracket (p \mapsto d) \rrbracket \mid (\nu q)(\llbracket (q \mapsto d) \rrbracket \mid \mathbf{reply}(q))) & \\
& \sim_d & (\nu q)(\llbracket (q \mapsto d) \rrbracket \mid \mathbf{reply}(q)) & (30)
\end{array}$$

The processes used in these relations are exactly those found in the proof of Th. 4.1, in the case of rule (gh red clone). Equations (27) and (29), for example, correspond to β-reductions, whereas (26) and (28) are deterministic communications, that is particular example of (11) and (12). In (30), we use the fact that p is only used inside a declaration to garbage collect the interpretation of the object $(p \mapsto d)$. The validity of (25) follows from the fact that $\gtrsim_d \subset \approx_d \subset \approx_b$.

This law can be viewed as a concurrent version of another equational law proved for the imperative $\boldsymbol{\varsigma}$ calculus in [22], namely:

$$(\mathbf{let}\, x = o \,\mathbf{in}\, \mathbf{clone}(x)) \;\approx\; o \tag{31}$$

Where o is the object $(\nu p)((p \mapsto [\, l_i = \varsigma(x_i)b_i{}^{i \in [1..n]}\,]) \Rsh p)$, the name p is not in $(\varsigma(x_i)b_i)_{i \in [1..n]}$, and $\approx$ is a contextual equivalence. Indeed, using (25), it is easy to prove that the interpretation of the terms in (31) are also barbed bisimilar in $\boldsymbol{\pi}^{\star}$.

We prove a second equational law, that is also true in the imperative calculus, that describe the behavior of a method invocation on a recently updated method. Namely we prove that:

$$\left\llbracket (\nu p)\begin{pmatrix} (p \mapsto d)\Rsh \\ \mathbf{let}\, x = (p \cdot l \Leftarrow \varsigma(y)b) \,\mathbf{in}\, x \cdot l \end{pmatrix} \right\rrbracket \approx_b \left\llbracket (\nu p)\begin{pmatrix} (p \mapsto d)\Rsh \\ \mathbf{let}\, y = (p \cdot l \Leftarrow \varsigma(y)b) \,\mathbf{in}\, b\{{}^{p}/_{y}\} \end{pmatrix} \right\rrbracket \tag{32}$$

Let P_1 be the term in the left part of (32), P_2 be the term in the right part, and let d' the denomination obtained from d by updating the method l in d with the body $\varsigma(y)b$. We have that:

$$\begin{aligned}
P_1 \;&\equiv\; (\nu p v)(\llbracket(p \mapsto d)\rrbracket \mid \langle v \Leftarrow (\lambda x)(x \cdot get_l)\rangle \mid (p \cdot put_l\,(\lambda y)\llbracket b\rrbracket\, v)) \\
&\gtrsim_d\; (\nu p v)(\llbracket(p \mapsto d')\rrbracket \mid \langle v \Leftarrow (\lambda x)(x \cdot get_l)\rangle \mid v p) && (33)\\
&\gtrsim_d\; (\nu p v)(\llbracket(p \mapsto d')\rrbracket \mid ((\lambda x)(x \cdot get_l))p) && (34)\\
&\gtrsim_d\; (\nu p)(\llbracket(p \mapsto d')\rrbracket \mid p \cdot get_l) && (35)\\
&\gtrsim_d\; (\nu p)(\llbracket(p \mapsto d')\rrbracket \mid \llbracket b\rrbracket\{^p/_y\}) && (36)\\
P_2 \;&\equiv\; (\nu p v)(\llbracket(p \mapsto d)\rrbracket \mid \langle v \Leftarrow (\lambda y)\llbracket b\rrbracket\rangle \mid (p \cdot put_l\,(\lambda y)\llbracket b\rrbracket\, v)) \\
&\gtrsim_d\; (\nu p)(\llbracket(p \mapsto d')\rrbracket \mid \llbracket b\rrbracket\{^p/_y\}) && (37)
\end{aligned}$$

The relation in (33), (36) and (37) are proved using the same reasoning than in the proof of (25). In particular we found exactly the same terms than in the proof of Th. 4.1, in the case of rule (gh red select) and (gh red updt). In (34), we use the law given in (12), while in (35) we use the fact that β-convertible terms are equivalent. Therefore we have $P_1 \approx_d ((\nu p)(\llbracket(p \mapsto d')\rrbracket \mid \llbracket b\rrbracket\{^p/_y\})) \approx_d P_2$ and, using the fact that $\approx_d \;\subset\; \approx_b$, we have that $P_1 \approx_b P_2$.

Finally, we prove the validity of the rule (struct let assoc), given in (17), in the particular case where the object a is a value, that is we prove that if $y \notin \mathbf{fn}(c)$, then:

$$\llbracket \mathbf{let}\, x = (\mathbf{let}\, y = u\, \mathbf{in}\, b)\, \mathbf{in}\, c\rrbracket \sim_b \llbracket \mathbf{let}\, y = u\, \mathbf{in}\, (\mathbf{let}\, x = b\, \mathbf{in}\, c)\rrbracket \qquad (38)$$

The proof of this relation is similar to the one of (32). Let v and w be two fresh names, we have:

$$\begin{aligned}
\llbracket \mathbf{let}\, x = (\mathbf{let}\, y = u\, \mathbf{in}\, b)\, \mathbf{in}\, c\rrbracket \;&=\; (\nu v)(\langle v \Leftarrow (\lambda x)\llbracket c\rrbracket\rangle \mid (\nu w)(\langle w \Leftarrow (\lambda y)\llbracket b\rrbracket\rangle \\
&\qquad\qquad\qquad\qquad\qquad \mid \mathbf{reply}(u)\, w)\, v) \\
&\gtrsim_d\; (\nu v w)(\langle v \Leftarrow (\lambda x)\llbracket c\rrbracket\rangle \mid \langle w \Leftarrow (\lambda y)\llbracket b\rrbracket\rangle \mid w\, u\, v) \\
&\gtrsim_d\; (\nu v w)(\langle v \Leftarrow (\lambda x)\llbracket c\rrbracket\rangle \mid \llbracket b\rrbracket\{^u/_y\}\, v)
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathbf{let}\, y = u\, \mathbf{in}\, (\mathbf{let}\, x = f\, \mathbf{in}\, c)\rrbracket \;&=\; (\nu w)(\langle w \Leftarrow (\lambda y)(\nu v)(\langle v \Leftarrow (\lambda x)\llbracket c\rrbracket\rangle \mid \llbracket b\rrbracket\, v)\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{reply}(u)\, w) \\
&\gtrsim_d\; (\nu w)(\langle w \Leftarrow (\lambda y)(\nu v)(\langle v \Leftarrow (\lambda x)\llbracket c\rrbracket\rangle \mid \llbracket b\rrbracket\, v)\rangle \mid w\, u) \\
&\gtrsim_d\; (\nu v w)(\langle v \Leftarrow (\lambda x)\llbracket c\rrbracket\rangle \mid \llbracket b\rrbracket\{^u/_y\}\, v)
\end{aligned}$$

We conjecture that the rule (struct let assoc) is valid in the more general case, in the sense that, if we add this rule to the definition of $\equiv$, then $a \equiv b$ implies that $\llbracket a\rrbracket \approx_b \llbracket b\rrbracket$. Nonetheless, as we said before, this rule is not necessary to faithfully interpret $\mathbf{conc}\boldsymbol{\varsigma}$.

6.2 Primitive for Synchronization

The calculus $\mathbf{conc}\boldsymbol{\varsigma}$ is concurrent, in the sense that multiple threads of computation can interact in parallel. Nonetheless, it is not obvious to understand how to synchronize these threads. The approach taken by the authors in [21], is to extend their object calculus with operators for mutexes. This new calculus, that is defined in Table 14, is denoted $\mathbf{conc}\boldsymbol{\varsigma}_m$.

Since the π-calculus is directly embedded in $\boldsymbol{\pi}^\star$, the target calculus of our interpretation embeds a natural notion of synchronization that is provided by communication. An example of this synchronization mechanism is given in the encoding of the call by value definition, *see* (4). We prove that our interpretation can be extended in a rather natural way to $\mathbf{conc}\varsigma_m$, and that this extension is sound with respect to the typing rules of $\mathbf{conc}\varsigma_m$ given in [21]. Moreover, mutexes appear (again) as a special kind of linearly (and recursively) defined ressources.

denotations:	d	$::=$	$\ldots$	as in Table 3
		$\mid$	locked	locked mutex
		$\mid$	unlocked	unlocked mutex
terms:	a, b, c	$::=$	$\ldots$	as in Table 3
		$\mid$	$\mathbf{acquire}(u)$	mutex acquisition
		$\mid$	$\mathbf{release}(u)$	mutex release

Table 14: Syntax of $\mathbf{conc}\varsigma_m$

The reduction in $\mathbf{conc}\varsigma_m$ is copied from the relation in $\mathbf{conc}\varsigma$. The structural congruence relation is defined as in Table 4, while the reduction relation is defined by the rules in Table 5, together with two new axioms for mutex acquisition and release (*see* Table 15).

$$\frac{}{(p \mapsto \text{unlocked}) \Rsh \mathbf{acquire}(p) \to (p \mapsto \text{unlocked}) \Rsh p} \ \text{(gh red acquire)}$$

$$\frac{(d \in \{\text{locked}, \text{unlocked}\})}{(p \mapsto d) \Rsh \mathbf{release}(p) \to (p \mapsto \text{unlocked}) \Rsh p} \ \text{(gh red release)}$$

Table 15: Reduction in $\mathbf{conc}\varsigma_m$

A mutex $(p \mapsto d)$, with $d \in \{\text{locked}, \text{unlocked}\}$, can be interpreted as a particular example of object with only two possible states, locked or unlocked. Informally, the encoding of a mutex is as a reference cell with two fields: *acquire* and *release*, and whose internal state can take only two values, that are modeled using two references: lock_p and unlock_p. Mutex release and acquisition are then interpreted as selection on the name p. That is:

$$[\![\mathbf{acquire}(p)]\!] = p \cdot \mathit{acquire} \qquad \text{and} \qquad [\![\mathbf{release}(p)]\!] = p \cdot \mathit{release}$$

The "recursive equations" verified by the two states lock_p and unlock_p are the following:

$$\left\{ \begin{array}{cccccccc} \text{in the state:} & \mathsf{unlock}_p & \text{selecting the method:} & \mathit{acquire} & \text{gives as result:} & \mathsf{lock}_p \mid \mathbf{reply}(p) \\ \ldots & \mathsf{unlock}_p & \ldots & \mathit{release} & \ldots & \mathsf{unlock}_p \mid \mathbf{reply}(p) \\ \ldots & \mathsf{lock}_p & \ldots & \mathit{acquire} & \ldots & \mathsf{lock}_p \mid p \cdot \mathit{acquire}, \\ \ldots & \mathsf{lock}_p & \ldots & \mathit{release} & \ldots & \mathsf{unlock}_p \mid \mathbf{reply}(p) \end{array} \right.$$

These equations can be directly translated in a blue calculus term using records and recursive declarations. Let $\mathbf{Mutex_p}$ denotes the following evaluation context:

$$\mathbf{Mutex_p} =_{\text{def}} (\nu \mathsf{lock}_p, \mathsf{unlock}_p) \left(\begin{array}{l} \langle \mathsf{unlock}_p {=} \langle p {\Leftarrow} \begin{bmatrix} \mathit{acquire} = \mathsf{lock}_p \mid \mathbf{reply}(p), \\ \mathit{release} \ = \mathsf{unlock}_p \mid \mathbf{reply}(p) \end{bmatrix} \rangle\rangle \\ \mid \langle \mathsf{lock}_p {=} \langle p {\Leftarrow} \begin{bmatrix} \mathit{acquire} = \mathsf{lock}_p \mid p \cdot \mathit{acquire}, \\ \mathit{release} \ = \mathsf{unlock}_p \mid \mathbf{reply}(p) \end{bmatrix} \rangle\rangle \\ \mid [\cdot] \end{array} \right)$$

The interpretation of $\mathbf{conc}\varsigma_m$ is given in Table. 16. We only give the encoding for the synchronization operators, the other cases are the same than in Table 6.

$$[\![\mathbf{acquire}(p)]\!] = p \cdot \mathit{acquire} \qquad [\![\mathbf{release}(p)]\!] = p \cdot \mathit{release}$$

$$[\![(p \mapsto \text{locked})]\!] = \mathbf{Mutex_p}[\mathsf{lock}_p] \qquad [\![(p \mapsto \text{unlocked})]\!] = \mathbf{Mutex_p}[\mathsf{unlock}_p]$$

Table 16: Interpretation of $\mathbf{conc}\varsigma_m$

It is easy to show that our interpretation of $\mathbf{conc}\varsigma_m$ is correct with respect to the reduction rules given in Table 15. The proof of this property is based on the following relations:

$$\begin{array}{rcl} [\![(p \mapsto \text{unlocked})]\!] \mid p \cdot \mathit{acquire} & \overset{*}{\to} & \mathbf{Mutex_p}[\begin{bmatrix} \mathit{acquire} = \mathsf{lock}_p \mid \mathbf{reply}(p), \\ \mathit{release} \ = \mathsf{unlock}_p \mid \mathbf{reply}(p) \end{bmatrix} \cdot \mathit{acquire}] \\ & \to & \mathbf{Mutex_p}[\mathsf{lock}_p \mid \mathbf{reply}(p)] \\ & \equiv & [\![(p \mapsto \text{locked}) \Rsh p]\!] \end{array}$$

and, if $d \in \{\text{locked}, \text{unlocked}\}$:

$$\begin{array}{rcl} [\![(p \mapsto d)]\!] \mid p \cdot \mathit{release} & \overset{*}{\to} & \mathbf{Mutex_p}[\begin{bmatrix} \mathit{acquire} = \ldots, \\ \mathit{release} \ = \mathsf{unlock}_p \mid \mathbf{reply}(p) \end{bmatrix} \cdot \mathit{release}] \\ & \to & \mathbf{Mutex_p}[\mathsf{unlock}_p \mid \mathbf{reply}(p)] \\ & \equiv & [\![(p \mapsto \text{unlocked}) \Rsh p]\!] \end{array}$$

Using these two relations and Theorem 4.1, it is easy to prove that our interpretation of $\mathbf{conc}\varsigma_m$ is complete.

Theorem 6.1. *If $a \to a'$ in $\boldsymbol{conc}\varsigma_m$, then $[\![a]\!] \overset{*}{\to} \sim_b [\![a']\!]$.*

Nonetheless, and contrary to the result obtained in Sect. 4, the converse properties does not hold. Indeed our interpretation of locked mutexes diverge when the field *acquire* is selected. That is we have:

$$\begin{array}{rcl} [\![(p \mapsto \text{locked})]\!] \mid p \cdot \mathit{acquire} & \overset{*}{\to} & \mathbf{Mutex_p}[\begin{bmatrix} \mathit{acquire} = \mathsf{lock}_p \mid p \cdot \mathit{acquire}, \\ \mathit{release} \ = \mathsf{unlock}_p \mid \mathbf{reply}(p) \end{bmatrix} \cdot \mathit{acquire}] \\ & \to & \mathbf{Mutex_p}[\mathsf{lock}_p \mid p \cdot \mathit{acquire}] \\ & \equiv & [\![(p \mapsto \text{locked})]\!] \mid p \cdot \mathit{acquire} \end{array}$$

Whereas the $\mathbf{conc}\varsigma_m$ term does not reduces.

As for the term of **concς**, our interpretation of mutexes is very natural and simple. It has also the nice property of being sound with respect to the type system, that is we can prove a result similar to Th. 5.3. Indeed, we can prove that the interpretation of mutexes in $\boldsymbol{\pi}^\star$ are well-typed terms and that the derived typing rules obtained for the synchronization primitives are equivalent to those defined by the authors in [21]. Let Mutex be the recursive type defined in Table 17. In particular, using type unfolding, it follows that:

$$\text{Mutex} \sim [\, \mathit{acquire} = \mathit{Reply}(\text{Mutex}), \mathit{release} = \mathit{Reply}(\text{Mutex}) \,]$$

Using this relation, and with the hypothesis that p is of type Mutex in the environment Γ, it is easy to show that:

$$\Gamma, \mathsf{unlock}_p : \mathit{Top}, \mathsf{lock}_p : \mathit{Top} \vdash \langle p \Leftarrow \begin{bmatrix} \mathit{acquire} = \mathsf{lock}_p \mid \mathbf{reply}(p), \\ \mathit{release} \;\, = \mathsf{unlock}_p \mid \mathbf{reply}(p) \end{bmatrix} \rangle : \mathit{Top}$$

and therefore: $\Gamma \vdash \mathbf{Mutex_p}[\mathsf{lock}_p] : \mathit{Top}$ and $\Gamma \vdash \mathbf{Mutex_p}[\mathsf{unlock}_p] : \mathit{Top}$. The rules derived for the synchronization primitives are given in Table 17.

$$\text{Mutex} =_{\text{def}} \mu\alpha.[\, \mathit{acquire} = \mathit{Reply}(\alpha), \mathit{release} = \mathit{Reply}(\alpha) \,]$$

$$\frac{\Gamma \vdash p : \text{Mutex} \qquad d \in \{\text{locked}, \text{unlocked}\}}{\Gamma \vdash \langle p \mapsto d \rangle : \mathit{Top}} \;\text{(type mutex)}$$

$$\frac{\Gamma \vdash u : \text{Mutex}}{\Gamma \vdash \mathbf{acquire}(u) : \mathit{Reply}(\text{Mutex})} \;\text{(type acquire)} \qquad \frac{\Gamma \vdash u : \text{Mutex}}{\Gamma \vdash \mathbf{release}(u) : \mathit{Reply}(\text{Mutex})} \;\text{(type release)}$$

Table 17: Interpretation of **concς**$_m$ Types

7 Conclusion and Related Work

We have shown that reduction and type judgements of **concς** can be derived in a rather simple and natural way in the blue calculus. Indeed, objects are modelled as a particular kind of declarations, $\langle p \mapsto d \rangle$, namely those that are linearly created each time they are accessed. This behaviour has to be compared with the consumable declaration, $\langle u \Leftarrow P \rangle$, used to model processes (of $\boldsymbol{\pi}$), and to the replicated and "immutable" declaration, $\langle u = P \rangle$, used to model functions.

Many theoretical studies address the problem of modelling object oriented languages in procedural languages [23], but few of them have succeeded to preserve powerful features such as subtyping. In [2], the authors propose a compositional interpretation of a typed object calculus with subtyping into $\mathtt{F}_{\leq\mu}$, a λ-calculus with second-order polymorphic types. Viswanathan improved this result in [40], where he gives a fully abstract interpretation in a first-order λ-calculus with references cells and records. In both solutions, the encoding relies on the so-called split method. Another interesting definition of a typed object calculus was given by Fisher and Mitchell [18]. But none of those calculi can model concurrent and interactive objects.

Jones [29] and Walker [41] have used the π-calculus for translating parallel object-oriented languages and for proving the validity of certain program transformations. But the source languages are untyped and rather simple. In [36], Sangiorgi gives the first interpretation of Abadi-Cardelli typed functional calculus with subtyping in π (a related work is [28]). This interpretation is extended to the imperative case in [30]. These interpretations, and the type system used, are very different from ours. For example, in the coding of method update, we do not use "relay constructs". Intuitively, in our encoding, the number of reductions when invoking a method does not depend on the number of method updates applied on the object. Another major difference is that, in the proof of the operational correctness property, we do not use a typed bisimulation, ie. we do not use information from the type system. There are also other formalisms used to model concurrent objects, mainly based on the π-calculus, such as [16, 19, 26, 33, 39], but we will not discuss them here.

Our work can be compared with the proposal of [40], where the author gives a syntax-oriented interpretation of a typed object calculus, and our approach brings the same benefits as his. In particular, our interpretation defines a type-safe way of implementing higher-order concurrent objects in the blue calculus, and therefore in $\boldsymbol{\pi}$. Another benefit of our encoding is that we validate some possible extensions of **conc$\boldsymbol{\varsigma}$**. The first possible extension is to add recursion and a notion of self-types to the type-system, and to add a new constant for the maximal types, say *Top*, that differs from the type given to processes (*Proc*). The most interesting extension that we consider is the addition of functions and higher-order constructs to **conc$\boldsymbol{\varsigma}$**. Indeed, functions can be coded in Abadi and Cardelli's object calculus, but to simulate the types of functions in a satisfactory way, they need to use universally and existentially quantified types to the detriment of type inference [1]. With our approach, we propose a natural extension of the object calculus with functions and this without noticeably modifying the definition of the equivalence or the type system, nor the "interesting" equational laws.

Acknowledgments

This work took place in the context of collaboration with Gérard Boudol on the blue calculus at INRIA Sophia-Antipolis. He has greatly influenced the present development. I want to thank Andrew Gordon and Paul Hankin for their helpful comments and an anonymous referee that pointed out that our encoding fails to capture the "usual" subtyping rule between objects with self types.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.
2. Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Proc. of POPL '96 – 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 396–409, 1996.
3. Roberto Amadio and Sanjiva Prasad. Localities and failures. Extended version from FST & TCS '94, 1995.
4. Hendrik Pieter Barendregt. *The Lambda Calculus, Its syntax and Semantics.* North Holland, 1981.
5. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
6. Michele Boreale, Cédric Fournet, and Cosimo Laneve. Bisimulations in the join-calculus. In D. Gries and W.P. de Roever, editors, *Proc. of PROCOMET '98 – Programming Concepts and Methods*, pages 68–86. Chapman & Hall, June 1998.
7. Gérard Boudol. The lambda-calculus with multiplicities. Technical Report 2025, INRIA, September 1993.
8. Gérard Boudol. The π-calculus in direct style. *Higher-Order and Symbolic Computation*, 11:177–208, 1998. Also appeared in Proc. of POPL '97, p. 228–241, January 1997.

9. Gérard Boudol and Silvano Dal-Zilio. An interpretation of extensible objects. In *Proc. of FCT '99 – 12th International Symposium on Fundamentals of Computation Theory*, August 1999.
10. Gérard Boudol and Cosimo Laneve. The discriminating power of multiplicities in the λ-calculus. Technical Report 2441, INRIA, December 1994.
11. Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. In *Proc. of TACS '97*, volume 1281 of *Lecture Notes in Computer Science*, pages 415–438. Springer-Verlag, 1997.
12. Luca Cardelli and John C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991.
13. Silvano Dal-Zilio. A bisimulation for the blue calculus. Technical Report 3664, INRIA, April 1999.
14. Silvano Dal-Zilio. *Calcul bleu: types et objets.* Thèse d'état, Université de Nice – Sophia-Antipolis, 1999. (forthcoming).
15. Luís Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.
16. Paolo Di Blasio and Kathleen Fisher. A calculus for concurrent objects. In *Proc. of CONCUR '96 – 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1996.
17. William Ferreira, Matthew Hennessy, and Alan Jeffrey. Combining typed λ-calculus with CCS. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Essays in Honour of Robin Milner*. MIT Press, 1998.
18. Kathleen Fisher and John C. Mitchell. A delegation-based object calculus with subtyping. In *proc. of FCT '95*, volume 965 of *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, 1995.
19. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. of POPL '96 – 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.
20. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus. In *Proc. of HLCL '98 – 3rd International Workshop on High-Level Concurrent Languages*, Elsevier ENTCS, 1998.
21. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. Technical Report 4XX, University of Cambridge Computer Laboratory, February 1999. Extended version of [20].
22. Andrew D. Gordon, Paul D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proc. of FST & TCS '97*, volume 1346 of *Lecture Notes in Computer Science*. Springer-Verlag, December 1997.
23. Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.
24. Paul Hankin. Personal communication, February 1999.
25. Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6:361–380, 1993.
26. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.
27. Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
28. Hans Hüttel and Josva Kleist. Objects as mobile processes. Technical Report RS-96-38, BRICS, October 1996. Presented at MFPS '96.
29. Cliff B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. of CONCUR '93 – 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.
30. Josva Kleist and Davide Sangiorgi. Imperative objects and mobile processes. In *Proc. of PROCOMET '98 – Working Conference on Programming Concepts and Methods*. North-Holland, 1998.
31. Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *Proc. of ICALP '98*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
32. Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.
33. Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proc. of TPPP '94 – Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer-Verlag, 1995.
34. Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
35. François Pottier. Symplifying subtyping constraints. In *Proc. of ICFP '96 – International Conference on Functional Programming*, pages 122–133, January 1996.
36. Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report 3000, INRIA, 1996.

37. Davide Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.
38. Davide Sangiorgi and Robin Milner. Techniques of "weak bisimulation up-to". In *Proc. of CONCUR '92 – 3rd International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.
39. Vasco T. Vasconcelos. Typed concurrent objects. In *Proc. of ECOOP '94 – 8th European Conference on Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1994.
40. Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proc. of LICS '98 – 13th Symposium on Logic in Computer Science*, pages 380–391, 1998.
41. David Walker. π-calculus semantics of object-oriented programming languages. In *Proc. of TACS '91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991.