An Interpretation of Typed Concurrent Objects in the Blue Calculus

Silvano Dal Zilio

Microsoft Research, Cambridge, England

Abstract. We propose an interpretation of a typed concurrent calculus of objects based on the imperative object calculus of Abadi and Cardelli. The target of our interpretation is a version of the blue calculus, a variant of the π-calculus that directly contains functions, with record and first-order types. We show that reductions and type judgments are derivable in a rather simple and natural way, and that our encoding can be extended to recursive and self-types, as well as to synchronization primitives. We also use our encoding to 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. However, 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 ς-calculus [1], in which the notion of object is primitive, and they have developed and studied type systems for this calculus.

In this paper, we give a model of concurrent object computation based on a modeling of objects as processes. We introduce some derived notations for objects and we give their translation in a version of the blue calculus, $\boldsymbol{\pi}^\star$ [5], extended with records. We type Blue calculus processes using an implicit, first-order type system based on the simply typed λ-calculus.

Using these derived constructs, we give an interpretation of a concurrent and imperative version of ς defined by Gordon and Hankin, **concς** [13]. We prove that this interpretation preserves reduction, typing and subtyping judgments. Therefore, our 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. Consequently, we obtain a type-safe way to implement higher-order concurrent objects in the Blue calculus, and therefore in the π-calculus ($\boldsymbol{\pi}$). Moreover, we can validate possible extensions of **concς** and, what is more original, we can use the embedding of **concς** in the Blue calculus to do equational reasoning on the source calculus. As an example, we sketch the proof of an equational law between objects at the end of this paper.

J. van Leeuwen et al. (Eds.): IFIP TCS 2000, LNCS 1872, pp. 409–424, 2000.

We organize the rest of the paper as follows. The next section introduces 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. Section 3 briefly introduces Gordon and Hankin's calculus of objects and gives its interpretation in $\boldsymbol{\pi}^\star$. We prove that **concς** is embedded in $\boldsymbol{\pi}^\star$ and that objects can be viewed as a particular kind of (linearly managed) resource. Section 4 is dedicated to the typing of processes and objects. It introduces a new type operator that is very well suited for typed continuation passing style transformations. Before concluding, we look at possible applications of our interpretation. Complete definition of the calculi and omitted proofs may be found in a long version of the paper [8].

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: $a_1, a_2, \ldots$, called names, 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 behaviors similar to those expressible in the π-calculus. This abstract machine sets up the foundation of the Blue calculus that can therefore be viewed, at the same time, as a concurrent λ-calculus and as an applicative π-calculus.

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, or store, 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 has an empty memory, denoted by the symbol ϵ, which can be extended with new *declarations* as in $(\mathcal{E} \mid \langle a_n = N \rangle)$. The stack has a similar structure and we use $(a_j, \mathcal{S})$ to denote the operation of adding the name a_j to the stack.

An execution of the functional AM starts in the initial configuration $\mathcal{K}_0$, with an empty stack and memory ($\mathcal{K}_0 \triangleq \{\epsilon; M; \epsilon\}$). The transition rules are defined as follows, where $M\{x \leftarrow a_j\}$ denotes the outcome of renaming all free occurrences of x in M to the name a_j.

$$\begin{aligned}
\{\mathcal{E}; a_j; \mathcal{S}\} &\to \{\mathcal{E}; N_j; \mathcal{S}\} && (\mathcal{E} = \cdots \mid \langle a_j \Leftarrow N_j \rangle \mid \cdots) \\
\{\mathcal{E}; \lambda x.M; (a_j, \mathcal{S})\} &\to \{\mathcal{E}; M\{x \leftarrow a_j\}; \mathcal{S}\} \\
\{\mathcal{E}; (MN); \mathcal{S}\} &\to \{(\mathcal{E} \mid \langle a_n = N \rangle); M; (a_n, \mathcal{S})\} && (a_n \text{ fresh name})
\end{aligned}$$

For example, to evaluate a function application we memorize the argument in a fresh memory location, and we add the name of this location to the stack.

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

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

Where the indices $i_j \in 1..n$ for each $j \in 1..k$. Each configuration corresponds to a λ-term and, for example, $\mathcal{K}_n$ corresponds to $(Ma_{i_1} \ldots a_{i_k})\{a_1 \leftarrow N_1\} \ldots \{a_n \leftarrow N_n\}$. Therefore, to each extension of the functional AM corresponds a generalization of the λ-calculus. In this section, we improve the functional AM until we obtain an execution model that compares to that of $\boldsymbol{\pi}$. The calculus defined by the extended AM is the Blue calculus.

We start with simple syntactical modifications. We modify our notations to use a sequence of applications instead of a stack, recasting the standard configuration $\mathcal{K}_n$ into: $\langle a_1 \ = \ N_1 \rangle \mid \cdots \mid \langle a_n \ = \ N_n \rangle \mid (Ma_{i_1} \ldots a_{i_k})$. With these modifications, we can reformulate the transition rules in the following way, with the side condition that the name a is fresh in rule (χ):

$$\begin{array}{rll} \langle a \ = \ N \rangle \mid \cdots \mid (aa_1 \ldots a_n) \ \rightarrow \ \langle a \ = \ N \rangle \mid \cdots \mid (Na_1 \ldots a_n) & & (\rho) \\ (\lambda x.M)a_1 \ldots a_n \ \rightarrow \ (M\{x \leftarrow a_1\})a_2 \ldots a_n & & (\beta) \\ (MN)a_1 \ldots a_n \ \rightarrow \ \langle a \ = \ N \rangle \mid Maa_1 \ldots a_n & & (\chi) \\ \mathcal{K} \rightarrow \mathcal{K}' \ \Rightarrow \ (\langle a \ = \ N \rangle \mid \mathcal{K}) \rightarrow (\langle a \ = \ N \rangle \mid \mathcal{K}') & & (\varpi) \end{array}$$

In this new presentation, rule (β) corresponds to a simplified form of beta-reduction, where we substitute a name, and not a term, for 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 synchronization, we use instead a particular kind of resource fetching.

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. The idea is to separate in each configuration, the store from the active part, that is, to separate the memory from the evaluated term. We allow some commutations though, with the restriction that the evaluated term is always at the right of the topmost parallel composition. More formally, we consider the following structural rules for parallel composition, where $P \leftrightarrows Q$ means that both $P \rightarrow Q$ and $Q \rightarrow P$ holds.

$$(M_1 \mid M_2) \mid M_3 \leftrightarrows M_1 \mid (M_2 \mid M_3) \qquad (M_1 \mid M_2) \mid M_3 \leftrightarrows (M_2 \mid M_1) \mid M_3$$

As a result, we obtain an asymmetric parallel composition operator, like the one defined in **concς**, or the formal description of CML [10]. Another consequence is that we can replace rule (ρ) by the simpler rule:

$$\langle a \ = \ N \rangle \mid (aa_1 \ldots a_n) \ \rightarrow \ \langle a \ = \ N \rangle \mid (Na_1 \ldots a_n) \tag{2.1}$$

Roughly speaking, we have transformed our functional AM to a Chemical AM (CHAM) in the style of [4]. 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:

$$(\langle a = N_1\rangle \mid \langle a = N_2\rangle \mid a) \to (\langle a = N_1\rangle \mid \langle a = N_2\rangle \mid N_1)$$
$$(\langle a = N_1\rangle \mid \langle a = N_2\rangle \mid a) \to (\langle a = N_1\rangle \mid \langle a = N_2\rangle \mid N_2)$$

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.

$$\langle a \Leftarrow N\rangle \mid (a a_1 \dots a_n) \to (N a_1 \dots a_n) \tag{2.2}$$

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 $a(x).P$ in $\boldsymbol{\pi}$, and thus it allows us 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.

Finally, we add the possibility to dynamically create fresh names. This mechanism is a distinctive feature of the π-calculus, and it is very easily implemented in our CHAM by adding the restriction operator, $(\nu a)\mathcal{K}$, together with new reduction rules. Using the restriction operator we can, for example, define the internal choice operator $(M \oplus N)$ to be the term $(\nu a)(\langle a \Leftarrow M\rangle \mid \langle a \Leftarrow N\rangle \mid a)$.

2.1 The Calculus

The Blue calculus can be viewed as the calculus obtained from the concurrent AM, in the same way that the join-calculus is derived from the reflexive CHAM defined in [12]. The following table gives the syntax of processes, P. The syntax depends on a set of atomic names, $\mathcal{N}$, ranged over by $a, b, \dots$ and partitioned in three kinds: variables $x, y, \dots$, bound by abstractions, $(\lambda x)P$; references $u, v, \dots$, bound by restrictions, $(\nu u)P$; labels $k, l, \dots$, used to name record fields.

Processes

$P, Q ::=$	process
a	name
$(\lambda x)P$	small λ-abstraction
(Pa)	application
$(P \mid Q)$	parallel composition
$(\nu u)P$	name restriction
$\langle u \Leftarrow P\rangle$	linear declaration
$\langle u = P\rangle$	replicated declaration
$[\,]$	empty record
$[P, l = Q]$	record extension
$(P \cdot l)$	selection

Our syntax enforces a restricted usage of names with respect to their kinds: we only allow declaration on references and abstraction on variables. This rule out terms such as $(\lambda x)\langle x \Leftarrow P\rangle$ for example.

The formal operational semantics of $\boldsymbol{\pi}^\star$ is given in a chemical style and defined using two relations. First, the *structural congruence* relation $\equiv$, which is equivalent to the relation $\leftrightarrows$ used previously, and that is used to rearrange terms. Second, the *reduction* relation, $\rightarrow$, which represents real computation steps, and that corresponds to (2.1), (2.2) and (β).

Structural congruence

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\lambda x)P \equiv (\lambda x)Q$	(Struct Lam)
$P \equiv Q \Rightarrow (Pa) \equiv (Qa)$	(Struct Appl)
$P \equiv Q \Rightarrow (P \cdot l) \equiv (Q \cdot l)$	(Struct Sel)
$P \equiv Q \Rightarrow (P \mid R) \equiv (Q \mid R)$	(Struct Par)
$P \equiv Q \Rightarrow (\nu u)P \equiv (\nu u)Q$	(Struct Res)
$P \equiv Q \Rightarrow \langle u \Leftarrow P\rangle \equiv \langle u \Leftarrow Q\rangle$	(Struct Decl)
$P \equiv Q \Rightarrow \langle u = P\rangle \equiv \langle u = Q\rangle$	(Struct Mdecl)
$P \equiv Q, R \equiv S \Rightarrow [P, l = R] \equiv [Q, l = S]$	(Struct Over)
$((P \mid Q) \mid R) \equiv (P \mid (Q \mid R))$	(Struct Par Assoc)
$((P \mid Q) \mid R) \equiv ((Q \mid P) \mid R)$	(Struct Par Comm)
$u \notin \mathbf{fn}(Q) \Rightarrow (\nu u)P \mid Q \equiv (\nu u)(P \mid Q)$	(Struct Res Par)
$u \notin \mathbf{fn}(Q) \Rightarrow Q \mid (\nu u)P \equiv (\nu u)(Q \mid P)$	(Struct Par Res)
$(\nu u)(\nu v)P \equiv (\nu v)(\nu u)P$	(Struct Res Res)
$(P \mid Q)a \equiv P \mid (Qa)$	(Struct Par Appl)
$(P \mid Q) \cdot l \equiv P \mid (Q \cdot l)$	(Struct Par Sel)
$a \neq u \Rightarrow ((\nu u)P)a \equiv (\nu u)(Pa)$	(Struct Res Appl)
$((\nu u)P) \cdot l \equiv (\nu u)(P \cdot l)$	(Struct Res Sel)

Reduction

$P \rightarrow P' \Rightarrow (Pa) \rightarrow (P'a)$	(Red Appl)
$P \rightarrow P' \Rightarrow (P \cdot l) \rightarrow (P' \cdot l)$	(Red Sel)
$P \rightarrow P' \Rightarrow (P \mid Q) \rightarrow (P' \mid Q)$	(Red Par 1)
$Q \rightarrow Q' \Rightarrow (P \mid Q) \rightarrow (P \mid Q')$	(Red Par 2)
$P \rightarrow P' \Rightarrow (\nu u)P \rightarrow (\nu u)P'$	(Red Res)
$P \rightarrow P', P \equiv Q \Rightarrow Q \rightarrow P'$	(Red $\equiv$)
$((\lambda u)P)v \rightarrow P\{u \leftarrow v\}$	(Red Beta)
$\langle u \Leftarrow P\rangle \mid (ua_1 \ldots a_n) \rightarrow (Pa_1 \ldots a_n)$	(Red Decl)
$\langle u = P\rangle \mid (ua_1 \ldots a_n) \rightarrow \langle u = P\rangle \mid (Pa_1 \ldots a_n)$	(Red Mdecl)
$[P, l = Q] \cdot l \rightarrow Q$	(Red Sel)
$k \neq l \Rightarrow [P, l = Q] \cdot k \rightarrow P \cdot k$	(Red Over)

Notations. We use $\tilde{a}$ to denote the sequence $a_1, \ldots, a_n$ and $\mathbf{fn}(P)$ to denote the set of free names in P. 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 extensions, $[\,[\,[\,]\,, l_1 = P_1\,]\,, \ldots, l_n = P_n\,]$, where $l_1, \ldots, l_n$ are pairwise distinct labels, into $[\,l_i = P_i{}^{i \in 1..n}\,]$, and a sequence of applications, $P a_1 \ldots a_n$, into $P\tilde{a}$.

2.2 Derived Operators

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

$$
\begin{array}{rcll}
\mathbf{def}\, u = P \,\mathbf{in}\, Q & \triangleq & (\nu u)(\langle u = P \rangle \mid Q) & \text{definition} \\
\mathbf{set}\, x = P \,\mathbf{in}\, Q & \triangleq & (\nu u)(\langle u \Leftarrow (\lambda x) Q \rangle \mid (Pu)) & \text{linear application} \\
\mathbf{reply}(a) & \triangleq & (\lambda r)(r\ a) & \text{synchronous message}
\end{array}
$$

We may interpret the first operator, subsequently called a definition, as an explicit substitution of P for the name u in Q, and we can define higher-order application, (PQ), as a shorthand for $\mathbf{def}\, u = Q \,\mathbf{in}\, (Pu)$, where $u \notin \mathbf{fn}(P) \cup \mathbf{fn}(Q)$. Note that the name u is recursively bound in $\mathbf{def}\, u = P \,\mathbf{in}\, Q$, and that it is possible to define recursion, $\mathbf{rec}\, u.P$, by $(\mathbf{def}\, u = P \,\mathbf{in}\, u)$. Using linear application, it is possible to define sequential composition, $P\,;\,Q$, as $(\mathbf{set}\, u = P \ \mathbf{in}\, Q)$, for some u not free in Q. The **reply** operator is used in continuation passing style encoding and, for example, to return a value in a linear application.

$$
\mathbf{set}\, x = \mathbf{reply}(a) \,\mathbf{in}\, Q \ \rightarrow\ (\nu u)(\langle u \Leftarrow (\lambda x) Q \rangle \mid (ua)) \ \stackrel{*}{\rightarrow}\ Q\{x \leftarrow a\}
$$

We can compare $\mathbf{reply}(a)$ with the *synchronous names* of the join-calculus, with the difference that, using our notation for higher-order application, it is possible to "reply" a general term, with $\mathbf{reply}(P)$ standing for the term $\mathbf{def}\, u = P \,\mathbf{in}\, (\lambda r)(ru)$.

3 Interpretation of the Concurrent Object Calculus

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

Expressions and results

$u, v ::=$	result
$\quad x$	variable
$\quad p$	name
$d ::=$	denotation
$\quad \{l_i = \varsigma(x_i) b_j{}^{i \in 1..n}\}$	

$a, b, c ::=$	term
u	result
$(p \mapsto d)$	denomination
$u{\cdot}l$	method invocation
$u{\cdot}l \Leftarrow \varsigma(x)b$	method update
$\mathbf{clone}(u)$	cloning
$\mathbf{let}\, x = a \,\mathbf{in}\, b$	sequencing
$(a \Rsh b)$	parallel composition
$(\nu p)a$	name restriction

The basic constructor of **concς** is the denomination, $(p \mapsto d)$, that, informally, adds a name to an object of $\boldsymbol{\varsigma}$ and acts like a special kind of declaration. It represents the store of an object-oriented program, like the declaration $\langle a = M \rangle$ represents the store of a configuration in the functional AM. We omit the formal definition of **concς** in this paper, but we hope that the reader unfamiliar with this calculus can grasp some idea of it from our interpretation. Its operational semantics is defined in a chemical style, with a structural equivalence, $\equiv$, analogous to the homonymous relation in $\boldsymbol{\pi}^\star$, and a reduction relation, $\rightarrow$. Some reductions of **concς** come from the **let** constructor, where values are names. For example, $(\mathbf{let}\, x = p \,\mathbf{in}\, b) \rightarrow b\{x \leftarrow p\}$. However, the basic interactions are between a denomination, and a method invocation, a method update or a cloning on its name. Assume d is the denotation $\{l_i = \varsigma(x_i)b_i{}^{i \in 1..n}\}$, we get that:

$$\frac{j \in 1..n}{(p \mapsto d) \Rsh p{\cdot}l_j \rightarrow (p \mapsto d) \Rsh b_j\{x_j \leftarrow p\}}$$

$$\frac{d' \triangleq \{l_i = \varsigma(x_i)b_i{}^{i \in 1..n, i \neq j}, l_j = \varsigma(x)b\} \qquad j \in 1..n}{(p \mapsto d) \Rsh (p{\cdot}l_j \Leftarrow \varsigma(x)b) \rightarrow (p \mapsto d') \Rsh p}$$

$$\frac{q \notin \mathbf{fn}(d)}{(p \mapsto d) \Rsh \mathbf{clone}(p) \rightarrow (p \mapsto d) \Rsh (\nu q)((q \mapsto d) \Rsh q)}$$

We interpret **concς** in the Blue calculus and we prove an operational correspondence result. In the process of defining the encoding of **concς**, denoted $[\![.]\!]$ hereafter, we will naturally introduce some derived operators for the object notations. We see that it allows regarding **concς** as embedded in the Blue calculus, and therefore $\boldsymbol{\pi}^\star$ as an object calculus.

We suppose that the **concς** names are included in $\boldsymbol{\pi}^\star$. Informally, the interpretation of a denomination $(p \mapsto d)$, where $d \triangleq \{l_i = \varsigma(x_i)b_i{}^{i \in 1..n}\}$, is a process modeling a "reference cell" that memorizes n values, $(\lambda x_1)[\![b_1]\!]$, ... , $(\lambda x_n)[\![b_n]\!]$. That is a recursively defined declaration of the name p, which encapsulates a record with $2n$ fields: the access field get_{l_i}, used to invoke method l_i; the field put_{l_i}, used to modify this method. We also add a field named *clone* that, when selected, creates a fresh cell with a copy of the current state. Schematically, we use the *split-method* technique of [2].

Let $\mathbf{R}(p, s, \tilde{x}, c)$ be the following record (of $\boldsymbol{\pi}^\star$):

$$\mathbf{R}(p, s, \tilde{x}, c) \triangleq \left[\begin{array}{l} \ldots \\ get_{l_i} = (s\tilde{x} \mid x_i p), \\ put_{l_i} = (\lambda y)(s x_1 \ldots x_{i-1}\, y\, x_{i+1} \ldots x_n \mid \mathbf{reply}(p)), \\ \ldots \\ clone = (s\tilde{x} \mid c\tilde{x}) \end{array}\right]^{i\in 1..n}$$

In our intuition, the identity of an object is a reference at which the object state can be fetched (the name p), its state is a record of methods as in the classical recursive records semantics [6] and encapsulation is naturally implemented using the **def** operator. In particular, the variable x_i is used to record the value of the method l_i, the name s is a pointer to a function that (given the x_i's) creates the object each time it is accessed, and the name c is a pointer to a function that creates a fresh copy of the object when it is cloned.

To encode a denomination, we encapsulate the record $\mathbf{R}(p, s, \tilde{x}, c)$ in a recursive definition that linearly manages a declaration of the name p. We define a notation for this definition.

$$\mathbf{Fobj}(p, \tilde{x}, c) \triangleq \mathbf{def}\ s = (\lambda \tilde{y})\langle p \Leftarrow \mathbf{R}(p, s, \tilde{y}, c)\rangle\ \mathbf{in}\ \langle p \Leftarrow \mathbf{R}(p, s, \tilde{x}, c)\rangle$$

We denote $\langle p \hookleftarrow \{\, l_i = (\lambda x_i) P_i^{\,i\in 1..n} \,\}\rangle$ the process that we obtain by binding the name c to the function that clones the object, and the names in $\tilde{x}$ to the functions $(\lambda x_1)P_1, \ldots, (\lambda x_n)P_n$.

$$\langle p \hookleftarrow \{l_i = (\lambda x_i)P_i^{\,i\in 1..n}\}\rangle \triangleq \left(\begin{array}{l} \mathbf{def}\ c = (\lambda \tilde{x})(\nu q)(\mathbf{Fobj}(q, \tilde{x}, c) \mid \mathbf{reply}(q)) \\ \mathbf{in\ def}\ u_1 = (\lambda x_1)P_1, \ldots, u_n = (\lambda x_n)P_n \\ \quad \mathbf{in}\ \mathbf{Fobj}(p, \tilde{u}, c) \end{array}\right)$$

Intuitively, the process $\langle p \hookleftarrow D\rangle$, where $D \triangleq \{l_i = (\lambda x_i)P_i^{\,i\in 1..n}\}$, can be divided into two components. An *active part*, the declaration $\langle p \Leftarrow \mathbf{R}(p, s, \tilde{x}, c)\rangle$, which can interact with other processes in parallel. A *passive part*, the recursive definitions on the names s, $\tilde{x}$ and c, which are used to memorize the internal state of the cell and to (linearly) recreate its active part each time the name p is invoked. Indeed, when $\langle p \hookleftarrow D\rangle$ interacts with the name p, the unique declaration on p is consumed and a unique output on the restricted name s, acting like a lock, is freed, which, in turn, frees a single declaration on p. Using the derived operator $\langle p \hookleftarrow D\rangle$, we give a very simple and direct interpretation of **concς**.

Translation rules

$[\![(p \mapsto \{l_i = \varsigma(x_i)b_i^{\,i\in 1..n}\})]\!] \triangleq \langle p \hookleftarrow \{\, l_i = (\lambda x_i)[\![b_i]\!]^{\,i\in 1..n} \,\}\rangle$
$[\![u]\!] \triangleq \mathbf{reply}(u)$
$[\![u{\cdot}l]\!] \triangleq (u \cdot get_l)$
$[\![u{\cdot}l \Leftarrow \varsigma(x)b]\!] \triangleq (u \cdot put_l\, (\lambda x)[\![b]\!])$

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

We can simplify this interpretation a step further by defining three shorthand for method select, method update and for cloning.

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

With these notations we can consider that **concς** is directly embedded in the Blue calculus. More interestingly, we embed a higher-order version of the object calculus, and it is possible to define terms that are not in **concς**, like $\mathbf{clone}(P \mid Q)$ for example, or the selector function $(\lambda x)(x \Leftarrow l)$. We can also derive a set of reduction sequences that simulate reduction in **concς**. Assume D is the association $\{\, l_i = (\lambda x_i)P_i^{\,i \in 1..n} \,\}$ and $j \in 1..n$ then:

$$\langle p \leftarrowtail D \rangle \mid p \Leftarrow l_j \;\xrightarrow{*}\; \langle p \leftarrowtail D \rangle \mid P_j\{x_j \leftarrow p\}$$

More formally, we prove that there is an operational correspondence between **concς** and the Blue calculus. To state this result, we use an observational equivalence between $\boldsymbol{\pi}^\star$-terms defined in [7], denoted $\approx$, that is a variant of weak barbed congruence [20]. Informally, this relation is the largest bisimulation that preserves simple observations called barbs and that is a congruence.

Theorem 3.1. *If $a \equiv b$, then $[\![a]\!] \equiv [\![b]\!]$. If $a \to a'$, then $[\![a]\!] \xrightarrow{*} \approx_b [\![a']\!]$. If $[\![a]\!] \to P$, then there exists a **concς**-term, a', such that $a \to a'$ and $P \xrightarrow{*} \approx_b [\![a']\!]$.*

4 Type System

We define a first-order type system for $\boldsymbol{\pi}^\star$, inspired by the (Curry-style) simply typed λ-calculus. It is essentially the type system given in [5], extended with subtyping, record types, recursion and a special type constructor for continuations, *Reply*(.). Then, we establish a set of derived typing rules for the object notations introduced in the previous section, that simulate the typing rules of **concς**.

We assume the existence of a denumerable set of type variables ranged over by $\alpha, \beta, \ldots$ The syntax of type expressions is given by the following grammar.

$$\begin{array}{lll} \tau, \vartheta, \varrho \;::=\; \alpha \mid (\tau \to \vartheta) \mid (\mu\alpha.\tau) \mid \mathit{Top} \mid & & \text{simple types} \\ \qquad [\,] \mid [\,\varrho\,,\, l : \tau\,] \mid \mathit{Reply}(\tau) & & \text{rows \& continuation type} \end{array}$$

We consider that types are well formed with respect to a simple kinding system, described in the extended version of this paper [8]. Informally, the kind system is used to constrain the type ϱ in the row $[\,\varrho\,,\, l : \tau\,]$ and, for example, to rule out types such as $[\,(\vartheta \to \varrho)\,,\, l : \tau\,]$.

In our system, a type environment is an association between names and types, and also between type variables and kinds: $\Gamma ::= \emptyset \mid \Gamma, a : \tau \mid \Gamma, \alpha :: \kappa$. The type system is based on four judgments: (1) $\Gamma \vdash \diamond$, and (2) $\Gamma \vdash \tau :: \kappa$, for well formed environment and types; (3) $\Gamma \vdash \tau <: \vartheta$, and (4) $\Gamma \vdash P : \tau$, given Γ, type τ is a subtype of ϑ and term P has type τ.

The type constructors are all borrowed from type systems for functional languages, apart from *Reply*(.) that is used to type the continuation operator **reply**(P) (and linear application) and that is described later. The type *Top* is the maximal type with respect to the subtyping relation. We make a non-standard use of this type constant: *Top* is used to type terms that may not be expected to return results, for example resources.

$$\frac{\Gamma \vdash P : \tau \qquad (u : \tau) \in \Gamma}{\Gamma \vdash \langle u \Leftarrow P \rangle : \mathit{Top}} \text{ (Proc Decl)} \qquad \frac{\Gamma \vdash P : \mathit{Top} \qquad \Gamma \vdash Q : \tau}{\Gamma \vdash (P \mid Q) : \tau} \text{ (Proc Par)}$$

Subtyping. We do not give the details of the subtyping rules here. The rules for the functional part of the system are standard. For example arrow types $(\tau \to \vartheta)$ are *contravariant* in the first parameter and *covariant* in the second. The subtyping rules for rows are less classical, and reflect the incremental construction of records. Provided the rows are well formed, we have the following subtyping rules, together with rules that allow identifying rows up-to reordering of their components.

$$\frac{}{\Gamma \vdash [\,\varrho\,,\, l : \tau\,] <: [\,]} \qquad \frac{\Gamma \vdash \varrho <: \varrho' \qquad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash [\,\varrho\,,\, l : \tau\,] <: [\,\varrho'\,,\, l : \tau'\,]}$$

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. The typing rules for the π-calculus operators are the rules (Proc Par) and (Proc Decl) defined previously. In particular, the type of a parallel composition, $P \mid Q$, is the type of the main thread of computation, which is Q. The typing rule for declarations, (Proc Decl), deserves more comment. Suppose that $\Gamma \vdash P : \vartheta$, with $(u : \tau) \in \Gamma$, and that u appears in subject position of a declaration $\langle u \Leftarrow Q \rangle$, for example $P \triangleq (\langle u \Leftarrow Q \rangle \mid R)$. Since we may substitute Q for an occurrence of u in R, see (2.2), the term Q must have the type τ. Using rule (Proc Decl), it is easy to derive typing rules for definitions and higher-order application, that are equivalent to those found in the ML type system.

$$\frac{\Gamma, u : \tau \vdash P : \tau \qquad \Gamma, u : \tau \vdash Q : \sigma}{\Gamma \vdash \mathbf{def}\, u = P \,\mathbf{in}\, Q : \sigma} \qquad \frac{\Gamma \vdash P : \tau \to \vartheta \qquad \Gamma \vdash Q : \tau}{\Gamma \vdash (P\,Q) : \vartheta}$$

Typing continuations. We explain the typing and subtyping rules for the operator *Reply*(.). Recall that **reply**(a) stands for $(\lambda r)(ra)$. Let α be a fresh type variable. It can be proved that if P has type τ, then $(\lambda r)(r\,P)$ has type $(\tau \to \alpha) \to \alpha$. In $(\tau \to \alpha) \to \alpha$, the variable α is implicitly quantified, in the sense that **reply**(P) can be given the type $\forall\alpha.((\tau \to \alpha) \to \alpha)$ in the ML type system. The type $(\tau \to \alpha) \to \alpha$ is often found in typed continuation passing style transformations, and in

λ-calculi with exceptions, where it is sometimes denoted $\neg^{\alpha}\neg^{\alpha}\tau$ [15]. To avoid the introduction of quantified types, we use a new operator to type the term $\mathbf{reply}(P)$, together with the introduction rule:

$$\frac{\Gamma \vdash P : \tau}{\Gamma \vdash \mathbf{reply}(P) : \mathit{Reply}(\tau)}$$

We can compare our usage of $\mathbf{reply}(.)$ with the usage of the operator **let** in ML, that can be defined as syntactic sugar for the term $(\lambda x.M)\,N$, but that is used in the type system to introduce parametric polymorphism.

It is possible to validate the two following rules using the interpretation of $\mathit{Reply}(\tau)$ as the type $(\tau \to \alpha) \to \alpha$ (for some fresh type variable α).

$$\frac{\Gamma \vdash P : \mathit{Reply}(\tau) \qquad \Gamma, x : \tau \vdash Q : \vartheta}{\Gamma \vdash \mathbf{set}\, x = P\ \mathbf{in}\, Q : \vartheta} \qquad \frac{\Gamma \vdash \tau <: \vartheta}{\Gamma \vdash \mathit{Reply}(\tau) <: \mathit{Reply}(\vartheta)}$$

The presence of $\mathit{Reply}(.)$ is only a minor extension to the traditional type system of $\boldsymbol{\pi}^{\star}$, and it does not modify its interesting properties. In particular, we prove that reduction preserves type judgments.

Theorem 4.1. *If $\Gamma \vdash P : \tau$ and $P \to P'$, then $\Gamma \vdash P' : \tau$.*

Typing objects. We prove a typed correspondence between $\mathbf{concs}\varsigma$ and the Blue calculus. To simplify our presentation, we first define a special notation for the type of a denomination. Apart from the use of the operator $\mathit{Reply}(.)$, this type is analogous to the one obtained in the encoding of Abadi-Cardelli functional object calculus given by Viswanathan [25] and Sangiorgi [23].

$$\mathrm{Obj}(\alpha.[\, l_i : \vartheta_i^{\,i\in 1..n}\,]) \triangleq \mu\alpha.\begin{bmatrix} \ldots & {\scriptstyle i\in 1..n} \\ \mathit{get}_{l_i} : \vartheta_i, & \\ \mathit{put}_{l_i} : (\alpha \to \vartheta_i) \to \mathit{Reply}(\alpha), & \\ \ldots & \\ \mathit{clone} : \mathit{Reply}(\alpha) & \end{bmatrix}$$

We prove that the object type, $\mathrm{Obj}(\alpha.\varrho)$, is the type of the name p in $\langle p \leftarrowtail D\rangle$. In $\mathrm{Obj}(\alpha.\varrho)$, the variable α is called the *self-type*. We simply write $\mathrm{Obj}(\varrho)$ if the self-type does not appear free in ϱ. We can give derived typing rules for the objects constructs defined in Sect. 3.

Derived typing rules for the embedding of objects

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

$$\frac{(p : \mathrm{A}) \in \Gamma \qquad \forall i \in 1..n \qquad \Gamma, x_i : \mathrm{A} \vdash P_i : \vartheta_i\{\alpha{\leftarrow}\mathrm{A}\}}{\Gamma \vdash \langle p \leftarrowtail \{\, l_i = (\lambda x_i)P_i^{\,i\in 1..n}\,\}\rangle : \mathit{Top}} \text{ (Proc Obj)}$$

$$\frac{\Gamma \vdash P : \mathrm{A} \qquad j \in 1..n \qquad \Gamma, x : \mathrm{A} \vdash Q : \vartheta_j\{\alpha{\leftarrow}\mathrm{A}\}}{\Gamma \vdash (P{\cdot}l_j \Leftarrow (\lambda x)Q) : \mathit{Reply}(\mathrm{A})} \text{ (Proc Updt)}$$

$$\frac{\Gamma \vdash P : \mathrm{A}}{\Gamma \vdash \mathbf{clone}(P) : \mathit{Reply}(\mathrm{A})} \text{ (Proc Clone)} \qquad \frac{\Gamma \vdash P : \mathrm{A} \qquad j \in 1..n}{\Gamma \vdash P \Leftarrow l_j : \vartheta_j\{\alpha{\leftarrow}\mathrm{A}\}} \text{ (Proc Invk)}$$

We give only the derivation for method update. Recall that $(P{\cdot}l_j \Leftarrow (\lambda x)Q)$ denotes the term $(P \cdot put_{l_j}\ (\lambda x)Q)$. Let A denotes the type $\mathrm{Obj}(\alpha.[\, l_i : \vartheta_i{}^{i\in 1..n}\,])$. Suppose that j is in $1..n$, that $\Gamma \vdash P : A$, and that $\Gamma, x : A \vdash Q : \vartheta_j\{\alpha{\leftarrow}A\}$. It follows that $\Gamma \vdash (\lambda x)Q : (\alpha \to \vartheta_j)\{\alpha{\leftarrow}A\}$, and that $\Gamma \vdash (P \cdot put_{l_j}) : ((\alpha \to \vartheta_j) \to \mathit{Reply}(\alpha))\{\alpha{\leftarrow}A\}$. Hence $\Gamma \vdash (P{\cdot}l_j \Leftarrow (\lambda x)Q) : \mathit{Reply}(A)$. Note that it is impossible to extend the object P with a new method, since the set $(put_{l_j})_{j\in 1..n}$ of fields is fixed. Moreover, like in Abadi-Cardelli calculus of first-order objects, it is impossible to refine the type of the updated method. Indeed, the type ϑ_j appears in contravariant position in field put_{l_j}, and in covariant position in field get_{l_j}. For the same reason, we can prove that object types are not covariant, that is $\varrho <: \sigma$ does not imply $\mathrm{Obj}(\varrho) <: \mathrm{Obj}(\sigma)$. However, we prove that width subtyping between object interfaces is sound.

Lemma 4.1. $\mathrm{Obj}([\, l_i : \vartheta_i{}^{i\in 1..n+m}\,]) <: \mathrm{Obj}([\, l_i : \vartheta_i{}^{i\in 1..n}\,])$.

The type system of **concς** is based on Abadi-Cardelli first-order object calculus, $\mathbf{Ob}_{1<:}$, extended with new type constants for expressions, processes and synchronization. In this system, a clear distinction is made between *expressions*, that is terms expected to return results, and *processes*, that intuitively represent stores of expressions. Then, the type system is used for two different goals. First, to guarantee that terms are well formed and that a name cannot be associated to two different denominations. Second, to avoid runtime errors, which are instances of the so-called "message not understood" problem. In this paper, we study a version that only guarantee safety of executions, but it is not difficult to extend our type system to accommodate the first requirement, as in type system ensuring the "unique receiver" property in $\boldsymbol{\pi}$ [3].

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. There is also a constant, *Proc*, used to type processes, like denominations for example. The type system is based on a subtyping relation, $E \vdash A <: B$, such that *Proc* is the maximal type and that $[\, l_i : A_i{}^{i\in 1..n+m}\,] <: [\, l_i : A_i{}^{i\in 1..n}\,]$.

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

Theorem 4.2. *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* $[\![E]\!] \vdash [\![a]\!] : \mathit{Reply}([\![A]\!])$.

In fact, we can prove a more general result than Theorem 4.2 since the type system for **concς** does not have recursive types, or self types, while our interpretation can capture such notions.

There is another modification to **concς** inspired by our interpretation. It consists in separating the two distinct roles of process and maximal type, that is to consider two different constants, *Top* and *Proc*, such that *Top* is the maximal

type and that *Proc* is used to type denominations. These two roles are collapsed in **concς**, as well as in the variant of $\boldsymbol{\pi}^\star$ defined in this paper. This emphasizes the fact that, in a parallel composition $(a \Rsh b)$, the value of a can never be communicated to the outside world, and thus only its side effects are observable. The fact that the value returned by a term is lost is not an example of a "runtime error", but we can consider that it is a programming mistake and, with our proposed modification, we can statically catch these mistakes.

5 Two Applications of our Interpretation

A first application is the interpretation of synchronization primitives. Although **concς** is a concurrent calculus, in the sense that multiple threads of computation can interact in parallel, it is not obvious how to synchronize these threads. The approach taken in [13] is to extend **concς** with operators for mutexes, that are defined as special kinds of denominations. The Blue calculus has a natural notion of synchronization based on asynchronous communication, exactly like in $\boldsymbol{\pi}$. Therefore, it is not surprising that our interpretation can be easily extended to model mutexes. What is more interesting is that our interpretation is also sound with respect to the typing rules for mutexes given in [13], and that mutexes appear (again) as a special kind of linearly defined resources.

A second application, that is the most original part of this work, is to prove equational laws between objects using barbed congruence between $\boldsymbol{\pi}^\star$-terms and our encoding. Let $\approx$ be the weak barbed congruence relation used in Theorem 3.1. We can use our interpretation to prove that two **concς**-terms are equivalent, by showing that their translations are equivalent. For example, if p is not free in d, we prove the following rules (among others):

$$[\![(\nu p)((p \mapsto d) \Rsh \mathbf{clone}(p))]\!] \approx [\![(\nu p)((p \mapsto d) \Rsh p)]\!]$$

$$\left[\!\!\left[(\nu p)\left(\begin{array}{l}(p \mapsto d)\Rsh\\ \mathbf{let}\, x = (p{\cdot}l \Leftarrow \varsigma(y)b)\,\mathbf{in}\, x{\cdot}l\end{array}\right)\right]\!\!\right] \approx \left[\!\!\left[(\nu p)\left(\begin{array}{l}(p \mapsto d)\Rsh\\ \mathbf{let}\, y = (p{\cdot}l \Leftarrow \varsigma(y)b)\,\mathbf{in}\, b\{y{\leftarrow}p\}\end{array}\right)\right]\!\!\right]$$

The first rule can be viewed as a concurrent version of an equational law proved for the imperative $\boldsymbol{\varsigma}$-calculus in [14], namely $(\mathbf{let}\, x = o\,\mathbf{in}\,\mathbf{clone}(x)) \approx o$, where o is the object $(\nu p)((p \mapsto d) \Rsh p)$, and p is not free in d.

An interesting fact is that the proofs of these equalities are very simple. Indeed, we only need to use "well-known" algebraic laws already proved for $\boldsymbol{\pi}^\star$ [7], like relation (5.1) below, similar to the replication theorem found in [19].

$$\mathbf{def}\, x = R\,\mathbf{in}\,(P \mid Q) \;\approx\; (\mathbf{def}\, x = R\,\mathbf{in}\,P) \mid (\mathbf{def}\, x = R\,\mathbf{in}\,Q) \qquad (5.1)$$

Another interesting fact is that the algebraic laws obtained on **concς** are still valid if one extends this calculus with new primitives that can be encoded in $\boldsymbol{\pi}^\star$, such as mutexes for example. Therefore, it is not necessary to modify the proof system each time the object calculus is extended.

As an example, we sketch the proof of the first equality. We use the notation of Sect. 3. Let $\mathbf{E_p}[.]$ be the context such that $[\![(p \mapsto d)]\!] = \mathbf{E_p}[\langle p \Leftarrow \mathbf{R}(p, s, \tilde{u}, c)\rangle]$.

$$
\begin{aligned}
[\![(\nu p)((p \mapsto d) \Rsh \mathbf{clone}(p))]\!] &\equiv (\nu p)\mathbf{E_p}[\langle p \Leftarrow \mathbf{R}(p, s, \tilde{u}, c)\rangle \mid p \cdot clone] && (1)\\
&\approx (\nu p)\mathbf{E_p}[\mathbf{R}(p, s, \tilde{u}, c) \cdot clone] && (2)\\
&\approx (\nu p)\mathbf{E_p}[s\tilde{u} \mid c\tilde{u}] && (3)\\
&\approx (\nu p)\mathbf{E_p}[s\tilde{u} \mid ((\lambda\tilde{x})(\nu q)(\mathbf{Fobj}(q, \tilde{x}, c) \mid \mathbf{reply}(q)))\tilde{u}] && (4)\\
&\approx (\nu p)\mathbf{E_p}[s\tilde{u} \mid (\nu q)(\mathbf{Fobj}(q, \tilde{u}, c) \mid \mathbf{reply}(q))] && (5)\\
&\approx (\nu p)(\mathbf{E_p}[s\tilde{u}] \mid (\nu q)([\![(q \mapsto d)]\!] \mid \mathbf{reply}(q))) && (6)\\
&\approx (\nu p)([\![(p \mapsto d)]\!] \mid (\nu q)([\![(q \mapsto d)]\!] \mid \mathbf{reply}(q))) && (7)\\
&\approx (\nu q)([\![(q \mapsto d)]\!] \mid [\![q]\!]) && (8)
\end{aligned}
$$

Step (1) uses an instance of the law: $(\nu u)(\langle u \Leftarrow P\rangle \mid u) \approx (\nu u)P$, and step (3) uses an instance of: $(\mathbf{def}\, x = R \,\mathbf{in}\, x) \approx (\mathbf{def}\, x = R \,\mathbf{in}\, R)$. In step (2) and (4), we use the fact that selection and β-reduction are deterministic reduction steps. For example we prove that $((\lambda x)P)a \approx P\{x{\leftarrow}a\}$. In step (5), we use (5.1) to distribute the definitions of $\mathbf{E_p}[.]$ over parallel composition, and in step (6) we use an intermediary result: $(\nu p)\mathbf{E_p}[s\tilde{u}] \approx (\nu p)[\![(p \mapsto d)]\!]$, that is implied by the laws used in step (3) and (4). In step (7), we use a "garbage collection" law similar to the following law: $((\nu u)\langle u \Leftarrow P\rangle) \mid Q \approx Q$.

6 Conclusion and Related Work

We have shown how to derive reduction and type judgments of **concς** in the Blue calculus in a rather simple and natural way. In our encoding, we model objects as a particular kind of declarations, $\langle p \leftarrowtail D\rangle$, that are "linearly managed". It is interesting to compare these declarations with the consumable declarations, $\langle u \Leftarrow P\rangle$, used to model processes (of the π-calculus), and with the replicated and immutable declarations, $\langle u = P\rangle$, used to model functions (of the λ-calculus).

Many theoretical studies address the problem of modeling object-oriented languages in procedural languages, but few of them have succeeded to preserve powerful features such as subtyping. In [2], the authors propose a compositional interpretation of a typed (sequential) object calculus with subtyping into $\mathtt{F}_{\leq\mu}$, a λ-calculus with second-order polymorphic types. Viswanathan improved this result in [25], where he gives a fully abstract interpretation in a first-order λ-calculus with reference cells and records. In both solutions, the encoding relies on the so-called split method. Fisher and Mitchell [11] proposed another interesting typed object calculus. However, none of those calculi can model concurrent and interactive objects.

In [23], Sangiorgi gives the first interpretation of Abadi-Cardelli typed functional calculus with subtyping in π (see also [17]). This interpretation is extended to the imperative case in [18, 21]. 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. Therefore, if these encoding were used to implement concurrent objects, we would provide a more efficient implementation of method invocation. Another major difference is that, in the proof of the operational correctness property, we do not use a typed bisimulation.

There are also other formalisms used to model concurrent objects, mainly based on the π-calculus, such as [9, 12, 16, 22, 24], that are not considered in this paper.

We can compare our work with the proposal of [25], where the author gives a syntax-oriented interpretation of a typed object calculus. 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 $\boldsymbol{\pi}$.

A benefit of our encoding is that we validate some possible extensions of **concς**, like the extension of the type system with recursive types and self-types, or the extension with a maximal types, say *Top*, that differs from the type given to processes. Another interesting extension considered in this paper is the addition of functions and higher-order constructs to **concς**. Indeed, functions can be coded in Abadi-Cardelli 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 without noticeably modifying the definition of the equivalence or the type system, nor the interesting equational laws. Another benefit is the study of equational laws between objects. We give an example of such equational laws at the end of Section 5. It would be interesting to study the equivalence obtained on **concς** using barbed congruence and our encoding.

Acknowledgments. This work took place in the context of collaboration with Gérard Boudol at INRIA Sophia-Antipolis. He has greatly influenced the present development. I had useful converstions with Andrew Gordon, Paul Hankin, Massimo Merro and Davide Sangiorgi.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. M. Abadi, L. Cardelli, and R. Viswanathan. An interpretation of objects and object types. In *Proc. of POPL '96*, pages 396–409, 1996.
3. R. Amadio and S. Prasad. Localities and failures. In *Proc. of FST & TCS '94*, volume 880 of *SLNCS*, pages 205–216, 1994.
4. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
5. G. Boudol. The π-calculus in direct style. *Higher-Order and Symbolic Computation*, 11:177–208, 1998. Also appeared in *Proc. of POPL '97*, Jan. 1997.

6. L. Cardelli and J. C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991.
7. S. Dal-Zilio. A bisimulation for the Blue calculus. TR 3664, INRIA, Apr. 1999.
8. S. Dal-Zilio. An interpretation of typed concurrent objects in the Blue calculus. Extended version, available at `http://research.microsoft.com/~sdal/`, 1999.
9. P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *Proc. of CONCUR '96*, volume 1119 of *LNCS*, Aug. 1996.
10. W. Ferreira, M. Hennessy, and A. Jeffrey. Combining typed λ-calculus with CCS. In *Essays in Honour of Robin Milner*. MIT Press, 1998.
11. K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. of FCT '95*, volume 965 of *LNCS*, pages 43–61, 1995.
12. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. of POPL '96*, pages 372–385, Jan. 1996.
13. A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *Proc. of HLCL '98*, Elsevier ENTCS, 1998.
14. A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proc. of FST & TCS '97*, volume 1346 of *LNCS*, Dec. 1997.
15. R. Harper and M. Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6:361–380, 1993.
16. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP '91*, volume 512 of *LNCS*, pages 133–147, 1991.
17. H. Hüttel and J. Kleist. Objects as mobile processes. TR RS-96-38, BRICS, Oct. 1996.
18. J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. of PROCOMET '98*. North-Holland, 1998.
19. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proc. of ICALP '98*, volume 1443 of *LNCS*, 1998.
20. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. of ICALP '92*, volume 623 of *LNCS*, pages 685–695, 1992.
21. U. Nestmann, H. Hüttel, J. Kleist, M. Merro. Aliasing Models for Object Migration. In *Proc. of Euro-Par '99*, volume 1685 of *LNCS*, pages 1353–1368, 1999.
22. B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Proc. of TPPP '94*, volume 907 of *LNCS*, pages 187–215, 1995.
23. D. Sangiorgi. An interpretation of typed objects into typed π-calculus. TR 3000, INRIA, 1996.
24. V. T. Vasconcelos. Typed concurrent objects. In *Proc. of ECOOP '94*, volume 821 of *LNCS*, pages 100–117, 1994.
25. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proc. of LICS '98*, pages 380–391, 1998.