Concurrent Objects in the Blue Calculus

Silvano Dal-Zilio*

INRIA Sophia-Antipolis

Abstract. We describe a model of concurrent objects based on the blue calculus ($\pi^\star$), a typed variant of the asynchronous π-calculus in which the notion of function is directly embedded. We propose a definition for a simple concurrent object-based calculus and show how objects can be translated in $\pi^\star$. We also present the type system for objects derived from our definition and we verify the expressiveness of the object calculus by giving a direct and adequate interpretation of Abadi and Cardelli object calculus: $\mathbf{Ob}_{1<:}$, that preserves subtyping.

1 Introduction

In his article [21], Milner states that the reduction relation of the π-calculus (π) is based on the *object paradigm*, in the sense that "what is transmitted and bound is never an object, but rather *access* to the object". There is a strong connection with object-oriented programming here: processes are objects (and states), and communication channels are the references used to name/access objects. But the object-oriented paradigm is more than computing with references. It deals with notions such as state encapsulation, dynamic dispatch and subtyping (or inheritance). The purpose of this paper is to give a model of concurrent object computation based on a modeling of "objects as processes". To this end, we introduce notations for object definition and we give their translations in the blue calculus [6] ($\pi^\star$) extended with records. Types for blue processes are given in a first-order type system with recursion and subtyping.

The expressiveness of our calculus is demonstrated by an interpretation of Abadi and Cardelli's typed functional calculus of objects: $\mathbf{Ob}_{1<:}$ [2], that preserves subtyping. This is an expected result, since Sangiorgi [24] has already given a translation from $\mathbf{Ob}_{1<:}$ to the π-calculus, but both the encoding and the type system used are very different. In particular, in our encoding, a method update does not create a new object but is rather modeled as a "change of state". Moreover, the operational correctness of our interpretation does not rely on information supplied by the type system and our encoding is direct (i.e. does not use continuations). Another difference is that we use imperative objects in our encoding, that is objects that can be cloned. These reflect differences in the goals. We are not focusing on equational rules on objects, but, rather, we try to define an expressive concurrent object calculus with a simple "implementation" in $\pi^\star$ and a type system based on a well established theory. Indeed, we believe that the best way to define a concurrent object calculus is to built it on top of an already studied process calculus. This method has several advantages, such as the possibility to reuse theoretical results and proof techniques. For example, in this paper, the type system for objects is not primitive and follows from the definition of the rather standard type system of the process calculus.

* Email: Silvano.Dal_Zilio@sophia.inria.fr. Address: INRIA Sophia-Antipolis, BP 93, 2004 route des Lucioles. F-06902 Sophia Antipolis cedex. Fax: (+33) 492.38.79.98

After the presentation of the blue calculus syntax, type system and operational equivalence, we introduce a simple concurrent object calculus in Sect. 3. In the remainder of this section, we show that the object operators can be derived in $\boldsymbol{\pi}^\star$ and we give the derived type system obtained for objects. In Sect. 4 we give an adequate encoding of $\mathbf{Ob}_{1<:}$ and we look at some examples of equivalences obtained using this encoding.

Related work Many theoretical studies address the problem of modeling object oriented languages in procedural languages [16, 1], but few of them have succeeded to preserve powerful features such as subtyping. In [3], the authors propose a compositional interpretation of a typed object calculus with subtyping into a functional calculus. But the target calculus used, $\mathbf{F}_{<:\mu}$, has second-order polymorphism, while the interpretation given in this paper uses a first-order type system. Very recently, R. Viswanathan [26] has proposed a fully abstract interpretation in a first-order λ-calculus with non-extensible records. Another interesting definition of a typed object calculus was given by Fisher and Mitchell [13]. But none of those calculi can model multiple interactive objects.

In the world of concurrency, Jones [18] and Walker [27] have used the π-calculus for translating parallel object-oriented languages and for proving the validity of certain program transformations. But the source languages studied are untyped and rather simple. In [24], Sangiorgi gives the first interpretation of the typed Abadi-Cardelli's calculus with subtyping in $\boldsymbol{\pi}$, and in [19], the authors give an interpretation of the imperative object calculus. There are also some works on the definition of concurrent calculus of objects obtained by extending sequential languages with concurrent operators: [12, 14].

2 The Calculus

The blue calculus is a variant of the mini asynchronous π-calculus [5] in which functions (abstractions, in the λ-calculus terminology) are directly embedded. Thus it has no choice, matching or guarded output operators. While $\boldsymbol{\pi}$ enforces an indirect style of programming, in the sense that one has to explicitly manage "result channels" to implement functions, $\boldsymbol{\pi}^\star$ provides a better "programming notation" for higher-order concurrency. Indeed Boudol shows in [6] that $\boldsymbol{\pi}$ is a "continuation passing style calculus".

The terms of $\boldsymbol{\pi}^\star$ (extended with records) are defined using three disjoint kinds of names. Variables: $x, y, z \cdots \in \mathcal{V}$, references: $u, v, w \cdots \in \mathcal{R}$ and labels: $k, l, m \cdots \in \mathcal{L}$. In the syntax, given in Table 1, D (in $\mathbf{def}\ D\ \mathbf{in}\ P$) is a sequence of definitions $x_1 = P_1, \ldots, x_n = P_n$, with the x_i's pairwise distinct[1].

Table 1 Syntax of the Blue Calculus: $\boldsymbol{\pi}^\star$

$$
\begin{array}{rcl@{\qquad}rcl}
a & ::= & x \mid u & P & ::= & a \mid (\boldsymbol{\lambda} x)P \mid (P\ a) \mid \mathtt{0} \mid (P \mid P) \mid (\boldsymbol{\nu}\, u)P \mid \\
 & & & & & \langle u \Leftarrow P \rangle \mid \mathbf{def}\ D\ \mathbf{in}\ P \mid [\,] \mid [\,l = P\,,\ P\,] \mid (P \cdot l)
\end{array}
$$

For convenience, we split the description of our calculus along three syntactical categories. For each of these cases, we examine the reduction relation ($\rightarrow$), the structural equiv-

[1] a definition can be empty, regarding "$\mathbf{def}\ \ \mathbf{in}\ P = P$" as identical to P

alence ($\equiv$) and the type system. The description uses the standard notion of *evaluation context*. Contexts are defined using the syntax of $\boldsymbol{\pi}^\star$-terms, plus a constant $[.]$. Evaluation contexts, $\mathrm{E}\,[]$, are contexts such that the hole does not occur within an abstraction, a record or a declaration:

$$\mathrm{E}\,[] ::= [.] \mid (\mathrm{E}\,[]\ a) \mid (\mathrm{E}\,[] \mid P) \mid (P \mid \mathrm{E}\,[]) \mid (\boldsymbol{\nu}\, u)(\mathrm{E}\,[]) \mid \mathbf{def}\ D\ \mathbf{in}\ \mathrm{E}\,[] \mid (\mathrm{E}\,[] \cdot l)$$

Two general rules can be given on the reduction relation:

$$P \to P' \wedge P \equiv Q \Rightarrow Q \to P' \qquad \text{and} \qquad P \to P' \Rightarrow \mathrm{E}\,[P] \to \mathrm{E}\,[P']$$

Concerning the structural equivalence, we assume that α-convertible processes are equal. The type system presented here is the Curry-style type system of [6], extended to support records, recursion and subtyping. In particular we have two type constants, $\bot$ and $\top$, that are, respectively, the smallest and the largest types:

$$\tau, \sigma, \omega \ldots ::= \bot \mid \top \mid \alpha \mid \mu\alpha.\tau \mid (\tau \to \tau) \mid [\,] \mid [\,l : \tau\,,\ \tau\,]$$

2.1 Description of the Functional Fragment of $\boldsymbol{\pi}^\star$

The functional part of our calculus is obtained using the constructs:

$$P, Q, R \ldots ::= \ldots \mid a \mid (\boldsymbol{\lambda} x)P \mid (P\ a) \mid \mathbf{def}\ D\ \mathbf{in}\ P$$

that is the "small" λ-calculus extended with the definition operator: $\mathbf{def}\ D\ \mathbf{in}\ P$. We use the term small since a process can only be applied to a name and not to another process: we say that the blue calculus is *name passing*. Nonetheless, the "high-order" λ-calculus application can be recovered using the definition:

$$(P\ Q) =_{\mathrm{def}} \mathbf{def}\ x = Q\ \mathbf{in}\ (P\ x) \qquad (x \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q))$$

There is a main difference here with respect to the original presentation of $\boldsymbol{\pi}^\star$ [6]. We have replaced "floating definitions" $\langle u = P \rangle$ (equivalent to an infinite parallel composition of "one-shot" declarations: $!\langle u \Leftarrow P \rangle$) by a construction that mixes together restriction, replication and definition: $\mathbf{def}\ x = R\ \mathbf{in}\ P =_{\mathrm{def}} (\boldsymbol{\nu} x)(\langle x = R \rangle \mid P)$.

We note by $\mathrm{def}(D)$ the set of variables defined by D, while $\mathrm{fn}(P)$ is the set of free names and variables in P. In the functional subset of $\boldsymbol{\pi}^\star$, binders are abstractions: $(\boldsymbol{\lambda} x)$ and definitions: $\mathbf{def}\ D\ \mathbf{in}\ P$. As usual, we write $(\boldsymbol{\lambda} x_1 \ldots x_n)P$ instead of $(\boldsymbol{\lambda} x_1) \ldots (\boldsymbol{\lambda} x_n)P$ and we use $\tilde{x}$ to denote the tuple of variables $x_1, \ldots, x_n$. Structural equivalence is defined by the following axioms [7]:

$$\begin{array}{rlrl}
(\mathbf{def}\ D\ \mathbf{in}\ P) \mid Q &\equiv \mathbf{def}\ D\ \mathbf{in}\ (P \mid Q) & \mathrm{def}(D) \cap \mathrm{fn}(Q) &= \emptyset \\
\mathbf{def}\ D\ \mathbf{in}\ (\mathbf{def}\ D'\ \mathbf{in}\ P) &\equiv \mathbf{def}\ D, D'\ \mathbf{in}\ P & \mathrm{def}(D') \cap \mathrm{fn}(D) &= \emptyset \\
\mathbf{def}\ D\ \mathbf{in}\ ((\boldsymbol{\nu}\, u)P) &\equiv (\boldsymbol{\nu}\, u)(\mathbf{def}\ D\ \mathbf{in}\ P) & u &\notin \mathrm{fn}(D) \\
(\mathbf{def}\ D\ \mathbf{in}\ P)\ a &\equiv \mathbf{def}\ D\ \mathbf{in}\ (P\ a) & a &\notin \mathrm{def}(D) \\
((\boldsymbol{\nu}\, u)P)\ a &\equiv (\boldsymbol{\nu}\, u)(P\ a) & a &\neq u
\end{array}$$

we define also "small" β reduction and the reduction rule for definitions:

$$((\boldsymbol{\lambda} x)P\ a) \to P\{a/x\}$$
$$\mathbf{def}\ D, x = R, D'\ \mathbf{in}\ \mathrm{E}\,[x\ a_1 \ldots a_n] \to \mathbf{def}\ D, x = R, D'\ \mathbf{in}\ \mathrm{E}\,[R\ a_1 \ldots a_n] \qquad (x \notin \mathrm{bn}(\mathrm{E}))$$

The typing rules for this fragment of our calculus are presented in Type System 1. Rule (def) uses the subtype relation ($<:$) defined in Sect. 2.3.

Type System 1 λ-Calculus Fragment

$$\frac{}{x:\tau,\Gamma\vdash x:\tau}\ \text{(taut)}\quad \frac{x:\tau,\Gamma\vdash P:\tau'}{\Gamma_{\restriction x}\vdash(\boldsymbol{\lambda}x)P:\tau\to\tau'}\ \text{(abs)}\quad \frac{\Gamma\vdash P:\tau\to\tau'\qquad\Gamma\vdash a:\tau}{\Gamma\vdash(P\ a):\tau'}\ \text{(app)}$$

$$\frac{\Gamma\cup\{x_i:\tau_i{}^{1\le i\le n}\}\vdash R_i:\tau_i\qquad\Gamma\cup\{x_i:\tau_i{}^{1\le i\le n}\}\vdash P:\omega}{\Gamma_{\restriction x_1,\dots x_n}\vdash\mathbf{def}\,x_1=R_1,\dots,x_n=R_n\,\mathbf{in}\,P:\omega}\ \text{(def)}$$

2.2 Description of the π-calculus Fragment of $\pi^\star$

In this section, we consider the operators directly derived from the π-calculus, that is:

$$P,Q,R\dots ::= \dots \mid \mathtt{0} \mid \langle u \Leftarrow P\rangle \mid (P \mid P) \mid (\boldsymbol{\nu}\,u)P$$

The new construct introduced here is the declaration: $\langle u \Leftarrow P\rangle$, that can be interpreted as a resource, located at u, accessible only once[2]. This construct is useful to model processes with a mutable state.

According to the conclusion of the original presentation of $\boldsymbol{\pi}^\star$ [6], the use of references is restricted in our calculus: although they can appear under a λ-abstraction, they can't be abstracted upon. For example, $(\boldsymbol{\lambda}u)\langle u \Leftarrow P\rangle$ is not a valid process. This restriction ensures that no new declaration on a given reference can be dynamically created, i.e. the well-known restriction that no receiver can be created on received names. Structural rules for this fragment are the "scope extrusion" rule of $\boldsymbol{\pi}$, the usual rules for the commutative monoid $(P, \mid, \mathtt{0})$ and rules managing application:

$$\begin{array}{cc} \mathtt{0}\mid P\equiv P & P\mid Q\equiv Q\mid P \\ P\mid(Q\mid R)\equiv(P\mid Q)\mid R & ((\boldsymbol{\nu}\,u)P)\mid Q\equiv(\boldsymbol{\nu}\,u)(P\mid Q)\ (\text{if } u\notin \text{fn}(Q)) \\ (P\mid Q)\ a\equiv(P\ a)\mid(Q\ a) & \langle u\Leftarrow P\rangle\ a\equiv\langle u\Leftarrow P\rangle \end{array}$$

Reduction consumes a declaration and fetches its definiendum at the output location:

$$\langle u \Leftarrow P\rangle \mid u\ a_1\dots a_n \to P\ a_1\dots a_n$$

Type System 2 π-Calculus Fragment

$$\frac{}{\Gamma\vdash\mathtt{0}:\bot}\ \text{(nil)}\quad \frac{\Gamma,u:\tau\vdash P:\tau}{\Gamma,u:\tau\vdash\langle u\Leftarrow P\rangle:\bot}\ \text{(decl)}$$

$$\frac{\Gamma\vdash P:\tau}{\Gamma_{\restriction x}\vdash(\boldsymbol{\nu}x)P:\tau}\ \text{(new)}\quad \frac{\Gamma\vdash P:\tau\qquad\Gamma\vdash Q:\tau}{\Gamma\vdash P\mid Q:\tau}\ \text{(par)}$$

[2] the declaration $\langle u \Leftarrow (\boldsymbol{\lambda}x)P\rangle$ is the equivalent of the π-calculus input guard $u(x).P$

2.3 Description of the Record Fragment of $\pi^\star$

Records are incrementally built from the empty record: $[\,]$, using the extend/update operation: $[\,l = P\,,\,Q\,]$ that adds/overrides the field l with value P to the record Q. Intuitively, a record is a function from a finite set of labels to values, and selection is function application. Therefore, the structural rules for records selection $(P \cdot l)$ are the same as the ones for application.

$$(P \mid Q)\cdot l \;\equiv\; (P\cdot l \mid Q\cdot l) \qquad \langle u \Leftarrow P\rangle \cdot l \;\equiv\; \langle u \Leftarrow P\rangle \qquad \cdots$$

We use $[\,l_1 = P_1, \ldots, l_n = P_n\,]$ instead of $[\,l_1 = P_1, \ldots, [\,l_n = P_n\,,\,[\,]\,]\,]$ whenever the l_i's are distinct. There are two reduction rules for records:

$$[\,l = P\,,\,Q\,]\cdot l \;\to\; P \quad \text{and} \quad [\,l = P\,,\,Q\,]\cdot k \;\to\; Q\cdot k \qquad (\text{if } k \neq l)$$

Our formalization of records is closely related to the one proposed by Wand [28], since there

Table 2 Subtyping Rules

$$\frac{\tau \sim \omega}{\tau <: \omega}\ (\text{sub axiom}) \qquad \frac{}{\bot <: \tau}\ (\text{sub bottom}) \qquad \frac{}{\tau <: \top}\ (\text{sub top})$$

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ (\text{sub trans}) \qquad \frac{\alpha <: \beta \Rightarrow \tau <: \omega}{\mu\alpha.\tau <: \mu\beta.\omega}\ (\text{sub rec})$$

$$\frac{\omega_1 <: \tau_1 \qquad \tau_2 <: \omega_2}{\tau_1 \to \tau_2 <: \omega_1 \to \omega_2}\ (\text{sub arrow}) \qquad \frac{\tau <: \tau' \qquad \omega <: \omega'}{[\,l : \omega\,,\,\tau\,] <: [\,l : \omega'\,,\,\tau'\,]}\ (\text{sub record})$$

is a single operation to either modify or add a field to a record. Thus, the definition of a maximal type (that types error processes, such as a "bad selection": $([\,]\cdot l)$) is crucial to prove subject reduction (and principal type if the type system is extended with polymorphism). Indeed, it allows the expression of "constraints" on types equivalent to those expressible with *row-variables* [23].

$$\begin{array}{lrcll}
(\text{error}) & [\,] & \sim & [\,l : \top\,,\,[\,]\,] & \\
(\text{swap}) & [\,l : \tau\,,\,[\,k : \omega\,,\,\upsilon\,]\,] & \sim & [\,k : \omega\,,\,[\,l : \tau\,,\,\upsilon\,]\,] & (\text{if } k \neq l)\\
(\text{crush}) & [\,l : \tau\,,\,[\,l : \omega\,,\,\upsilon\,]\,] & \sim & [\,l : \tau\,,\,\upsilon\,] &
\end{array}$$

For example, the restriction operation $P\backslash l$ of [8], that removes a field of label l from P, can be

Type System 3 Record-Calculus Fragment

$$\frac{}{\Gamma \vdash [\,] : [\,]}\ (\text{void}) \qquad \frac{\Gamma \vdash P : \tau \qquad \Gamma \vdash Q : \omega}{\Gamma \vdash [\,l = P\,,\,Q\,] : [\,l : \tau\,,\,\omega\,]}\ (\text{record})$$

$$\frac{\Gamma \vdash P : [\,l : \tau\,,\,\omega\,]}{\Gamma \vdash (P\cdot l) : \tau}\ (\text{selection}) \qquad \frac{\Gamma \vdash P : \tau \qquad \tau <: \omega}{\Gamma \vdash P : \omega}\ (\text{subt})$$

coded in $\boldsymbol{\pi}^\star$ with $[\, l = \mathfrak{e}\,,\ P\,]$ (and $\mathfrak{e}$ a constant of type $\top$). We also define an equivalence over types, $\sigma \sim \omega$, that is a congruence with three axioms. In particular one can prove, using rule (error), that $[\,] \sim [\, l_1 : \top, \ldots, l_n : \top\,]$. Rules for subtyping are rather standard. In particular, the usual notion of width subtyping can be derived in our system.

Proposition 1 (Width subtyping). $[\, l_1 : \tau_1, \ldots, l_{n+m} : \tau_{n+m}\,] <: [\, l_1 : \tau_1, \ldots, l_n : \tau_n\,]$

The proof uses the rules (sub record), (swap) and (error). For example $[\, k : \tau, l : \sigma\,] <: [\, l : \sigma\,]$ is a consequence of: $[\, k : \tau\,,\ [\, l : \sigma\,]\,] <: [\, k : \top\,,\ [\, l : \sigma\,]\,] \sim [\, l : \sigma, [\, k : \top\,]\,] \sim [\, l : \sigma\,]$. We also have the standard result of subject reduction.

Proposition 2 (subject reduction for Type System $1 + 2 + 3$). *If $\Gamma \vdash P : \tau$ and $P \to P'$, then $\Gamma \vdash P' : \tau$.*

2.4 Operational Equivalence

We define a relation of observational equivalence between $\boldsymbol{\pi}^\star$-terms ($\approx$) used to prove the correctness of our interpretation of $\mathbf{Ob}_{1<:}$ (Theorem 6). This relation is the biggest bisimulation that preserves simple observations called barbs and that is a congruence [17] (is is a variant of the weak barbed congruence [20]). But, whereas the observable behaviours considered in CCS and $\boldsymbol{\pi}$ are the visible outputs[3], we choose instead to observe the presence of values, i.e. abstractions, as in the definition of traditional Morris-style equivalences in the λ-calculus [4, ex. $16.5.5$]. We say that the process P is a value (is observable) if it is structurally equivalent to $(\boldsymbol{\nu}\, \tilde{u})((\boldsymbol{\lambda} x)V \mid R)$. This is denoted $P \downarrow$. The weak version of barbs used in the definition of $\approx$ is $P \Downarrow =_{\mathrm{def}} \exists V, P \to^* V \downarrow$.

Definition 3. A relation S is a *weak barbed simulation* if for each $(P, Q) \in S$, (1) : whenever $P \to P'$ then $Q \to^* Q'$ and $(P', Q') \in S$; $(2) : P \Downarrow$ implies $Q \Downarrow$. S is a (weak) barbed bisimulation if S and S^{-1} are (weak) barbed simulation. P and Q are observationally equivalent, written $P \approx Q$, iff $(P, Q) \in S$ for some weak barbed bisimulation S that is also a congruence.

3 Modeling Objects in the Blue Calculus

Before to introduce the specification of the object constructs and their translation in $\boldsymbol{\pi}^\star$, we present the example of the prototype of all objects: the *mutable cell*. Indeed, in our intuition, the object identity is the reference at which the object state can be fetched, its state is a record of methods (as in the classical recursive records semantic [8]) and encapsulation is naturally implemented using mutable cells. Let $\mathrm{R}_o(b)$ denotes the record:

$$\mathrm{R}_o(b) \ =_{\mathrm{def}}\ \big[\, get = (o\ b \mid b),\ put = (\boldsymbol{\lambda} x)(o\ x)\,\big]$$

The cell process with "name " O is defined by:

$$\mathrm{CELL}(O) \ =_{\mathrm{def}}\ \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle\, O \Leftarrow \mathrm{R}_o(b)\rangle\, \mathbf{in}\, o$$

[3] this is equivalent to observe free names in head position in $\boldsymbol{\pi}^\star$

Application: $(\text{CELL}(O)\ a_0)$, initializes the cell to the value a_0. It is easy to see that[4] $(\text{CELL}\ a_0 \mid O \cdot get)$ and $(\text{CELL}\ a_0 \mid O \cdot put\ a)$ evaluate in a deterministic way:

$$\begin{aligned}(\text{CELL}\ a_0) \mid (O \cdot get) &\to \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (\langle O \Leftarrow \mathrm{R}_o(a_0)\rangle \mid O \cdot get)\\ &\to \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (\mathrm{R}_o(a_0) \cdot get)\\ &\to \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (o\ a_0 \mid a_0) \ \equiv\ (\text{CELL}\ a_0) \mid a_0\end{aligned}$$

$$(\text{CELL}\ a_0) \mid (O \cdot put\ a) \to^* \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (o\ a) \ \equiv\ (\text{CELL}\ a)$$

It is interesting to notice the linear use of the reference O in $(\text{CELL}(O)\ a)$. If the cell is invoked, we consume the unique declaration $\langle O \Leftarrow \mathrm{R}_o(a)\rangle$. Thus, a unique message $(o\ a')$, acting like a lock, is freed in the evaluation process, which, in turn, frees a single declaration $\langle O \Leftarrow \mathrm{R}_o(a')\rangle$. Thus, we have the invariant that there is exactly one resource available at address O, and that this resource keeps the last value passed in a $(O \cdot put)$ call.

We give two generalizations of the mutable cell that will be useful to define the processes modeling objects. First, we define a n-ary cell, i.e. a cell that memorizes n different values. This cell uses a record with $2n$ fields: $put_1, \ldots, put_n$ and $get_1, \ldots, get_n$ instead of $\mathrm{R}_o(b)$:

$$\mathrm{R}_o(b_1, \ldots, b_n) \ =_{\text{def}}\ \left[\begin{array}{ll}\ldots & {\scriptstyle 1\le i\le n}\\ get_i = (o\ b_1 \ldots b_n \mid b_i), & \\ put_i = (\boldsymbol{\lambda} x_i)(o\ b_1 \ldots x_i \ldots b_n), & \\ \ldots & \end{array}\right]$$

$$\text{NCELL}(O) \ =_{\text{def}}\ \mathbf{def}\, o = (\boldsymbol{\lambda} b_1 \ldots b_n)\langle O \Leftarrow \mathrm{R}_o(b_1, \ldots, b_n)\rangle \,\mathbf{in}\, o$$

$$(\text{NCELL}(O)\ a_1 \ldots a_n) \mid O \cdot get_i \to^* (\text{NCELL}(O)\ a_1 \ldots a_n) \mid a_i$$

It is also possible to define a cloneable cell. To this end, we extend, in CELL, the record $\mathrm{R}_o(b)$ with a new field $clone$ and we add a recursive definition around this new cell definition. A selection on the field $clone$ produces a fresh cell memorizing a copy of its current value:

$$\mathrm{S}_o(b) \ =_{\text{def}}\ [\, clone = (o\ b \mid x_{clone}\ b)\,,\ \mathrm{R}_o(b)\,] \qquad \text{CELL}(O) \ =_{\text{def}}\ \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{S}_o(b)\rangle \,\mathbf{in}\, o$$

$$\text{CCELL}(O) \ =_{\text{def}}\ \mathbf{def}\, x_{clone} = (\boldsymbol{\lambda} b)(\boldsymbol{\nu} A)((\text{CELL}(A)\ b) \mid A) \,\mathbf{in}\, \text{CELL}(O)$$

$$\begin{aligned}(\text{CCELL}(O)\ a_0) \mid O \cdot clone &\to^* \mathbf{def}\, x_{clone} = \ldots \,\mathbf{in}\, (\text{CELL}(O)\ a_0) \mid (x_{clone}\ a_0)\\ &\to^* \mathbf{def}\, x_{clone} = \ldots \,\mathbf{in}\, (\text{CELL}(O)\ a_0) \mid (\boldsymbol{\nu} A)(\text{CELL}(A)\ a_0 \mid A)\end{aligned}$$

3.1 A Concurrent Calculus of Objects

In order to handle more elaborate objects than the (canonical) examples of mutable cells, we introduce a calculus of concurrent objects by specifying a set of operators and their operational semantics. In the remainder of the section, we prove that these operators (and their associated reduction rules) are derived from $\pi^\star$. We study also the derived types given to objects. In the specification of this calculus (Table 3), we distinguish a subset $\mathcal{O}$ of references (which we call objects names, $O, A, B, \cdots \in \mathcal{O}$) and we use L to denote an "object body": $L = l_1 = \varsigma(x_1)P_1, \ldots, l_n = \varsigma(x_n)P_n$.

[4] to simplify the examples, we use CELL to denote CELL(O)

Table 3 Specification of Operators and Reduction Rules for Objects in $\pi^\star$

$\varsigma(x)P$	method with self parameter x and body P
$\mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i)P_i^{\,1\le i\le n} \right\} \mathbf{in}\ P$	object with n methods $l_1, \ldots, l_n$
$P \Leftarrow l$	invocation of method l
$P \leftarrow l = \varsigma(x)Q$	update of method l with body $\varsigma(x)P$
$\mathbf{clone}(O)$	cloning of object O

Let $L =_{\mathrm{def}} l_1 = \varsigma(x_1)P_1, \ldots, l_n = \varsigma(x_n)P_n$

$$\mathbf{obj}\, O = \{L\} \mathbf{in}\, \mathrm{E}\,[O \Leftarrow l_j] \ \rightarrow_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{L\} \mathbf{in}\, \mathrm{E}\,[P_j\{O/x_j\}]$$

$$\mathbf{obj}\, O = \{L\} \mathbf{in}\, \mathrm{E}\,[O \leftarrow l_j = \varsigma(x)P] \rightarrow_{\pi^\star_\varsigma} \mathbf{obj}\, O = \{l_j = \varsigma(x)P, l_i = \varsigma(x_i)P_i^{\,i\neq j}\} \mathbf{in}\, \mathrm{E}\,[O]$$

$$\mathbf{obj}\, O = \{L\}\, \mathbf{in}\, \mathrm{E}\,[\mathbf{clone}(O)] \rightarrow_{\pi^\star_\varsigma} \mathbf{obj}\, O = \{L\}\, \mathbf{in}\, (\mathbf{obj}\, A = \{L\}\, \mathbf{in}\, \mathrm{E}\,[A]) \quad (A \notin \mathrm{bn}(\mathrm{E}))$$

An example of object is the one that produces an infinite copy of itself. Let L_∞ be the body: $l = \varsigma(x)(\mathbf{clone}(x) \mid x \Leftarrow l)$, then:

$$\mathbf{obj}\, O = \{L_\infty\}\, \mathbf{in}\, O \Leftarrow l \ \rightarrow^*_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{L_\infty\}\, \mathbf{in}\, (\mathbf{obj}\, A = \{L_\infty\}\, \mathbf{in}\, (A \mid O \Leftarrow l)) \ \rightarrow^*_{\pi^\star_\varsigma}\ \ldots$$

3.2 Interpretation of the Derived Object Constructs

Processes modeling objects are inspired from the encoding of the mutable cell. In the definition given in Table 4, an object ($\mathbf{obj}\, O = \{ l_i = \varsigma(x_i)P_i^{\,1\le i\le n}\} \mathbf{in}\ P$), is a n-ary cell with a field $clone$ to allow object cloning and $2n$ fields to allow method invocation (get_{l_i}) and method update (put_{l_i}). In this definition, a method $\varsigma(x)P$ is an abstraction $(\boldsymbol{\lambda} x)P$. This function, also called premethod, has one argument: the name of the current object (also called the self-parameter). Note that we restrict the scope of an object name to the definition of the object it refers to, and that method update returns "a reference" to the modified object. This is almost the behaviour of $\mathbf{Ob}_{1<:}$ (see Table 5),

$$\mathrm{T}_o(O, b_1, \ldots, b_n) \ =_{\mathrm{def}}\ \left[\begin{array}{l} \ldots \\ get_{l_i} = (o\ b_1 \ldots b_n \mid b_i\ O), \\ put_{l_i} = (\boldsymbol{\lambda} x_i)(o\ b_1 \ldots x_i \ldots b_n \mid O), \\ \ldots \\ clone = (o\ b_1 \ldots b_n \mid x_{clone}\ b_1 \ldots b_n) \end{array}\right]$$

$$\mathrm{OBJ}(O) \ =_{\mathrm{def}}\ \mathbf{def}\ o = (\boldsymbol{\lambda} b_1 \ldots b_n)\langle O \Leftarrow \mathrm{T}_o(O, b_1, \ldots, b_n)\rangle\ \mathbf{in}\ o$$

Another remark is that we use only field selection and application in the definition of cloning, method invocation and method update. Thus the definition of structural equivalence allows us, for example, to derive the following laws, showing that these (derived) operators acts like application:

$$(\mathbf{def}\, D\, \mathbf{in}\, P \mid Q) \Leftarrow l \ \equiv\ \mathbf{def}\, D\, \mathbf{in}\, (P \Leftarrow l \mid Q \Leftarrow l) \qquad (\langle u \Leftarrow P\rangle \Leftarrow l) \ \equiv\ \langle u \Leftarrow P\rangle$$

Table 4 Definition of the Derived Operators for Objects

$$P \Leftarrow\!\!\!\!\!\!\Rightarrow l =_{\rm def} (P \cdot get_l) \quad \mathbf{clone}(O) =_{\rm def} (O \cdot clone) \quad P \leftarrow l = \varsigma(x)Q =_{\rm def} (P \cdot put_l\ (\boldsymbol{\lambda} x)Q)$$

$$\mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i) P_i^{\,1 \le i \le n} \right\} \mathbf{in}\ P =_{\rm def} \begin{cases} \mathbf{def}\ x_{clone} = (\boldsymbol{\lambda} f_1 \ldots f_n)(\boldsymbol{\nu} A)(\text{OBJ}(A)\ f_1 \ldots f_n \mid A) \\ \mathbf{in}\ (\boldsymbol{\nu} O)((\text{OBJ}(O)\ (\boldsymbol{\lambda} x_1) P_1 \ldots (\boldsymbol{\lambda} x_n) P_n) \mid P) \end{cases}$$

The next result states that there is an operational correspondence between $\rightarrow_{\pi^\star_\varsigma}$ (defined in Table 3) and $\rightarrow$. These properties are proved using a simple induction.

Theorem 4 (Operational Equivalence). *The specification of the object reduction rules is* complete *with respect to the encoding of objects in $\pi^\star$: $P \rightarrow_{\pi^\star_\varsigma} P' \Rightarrow P \rightarrow^* P'$. It is also* sound*: $P \rightarrow Q$ implies that $Q \rightarrow^* Q'$ with $P \rightarrow_{\pi^\star_\varsigma} Q'$.*

3.3 Derived Type System for Objects

We present now the derived type system for objects obtained using Type System $1 + 2 + 3$. We first define an abbreviation: $(\mathbf{obj}\ \rho)$, for the type of $\mathrm{T}_o(O, b_1, \ldots, b_n)$. It can be proved that:

$$\begin{pmatrix} O : \tau,\ b_1 : (\tau \rightarrow \rho_1), \ldots, b_n : (\tau \rightarrow \rho_n), \\ o : (\tau \rightarrow \rho_1) \rightarrow \cdots \rightarrow (\tau \rightarrow \rho_n) \rightarrow \bot, \\ x_{clone} : (\tau \rightarrow \rho_1) \rightarrow \cdots \rightarrow (\tau \rightarrow \rho_n) \rightarrow \tau \end{pmatrix} \vdash \mathrm{T}_o(O, b_1, \ldots, b_n) : \left[\begin{array}{l} {\scriptstyle 1 \le i \le n} \\ get_{l_i} = \rho_i, \\ put_{l_i} = (\tau \rightarrow \rho_i) \rightarrow \tau, \\ clone = \tau \end{array} \right]$$

Thus the type of an object, which is also the type of its object name, O, is the recursive type:

$$(\mathbf{obj}\ [\, l_i : \rho_i^{\,1 \le i \le n}\,]) =_{\rm def} \mu t. \left[\begin{array}{l} {\scriptstyle 1 \le i \le n} \\ get_{l_i} = \rho_i, \\ put_{l_i} = (t \rightarrow \rho_i) \rightarrow t, \\ clone = t \end{array} \right]$$

A first remark is that an object type is not covariant. That is $\rho <: \sigma$ does not implies $(\mathbf{obj}\ \rho) <:$ $(\mathbf{obj}\ \sigma)$. Indeed, the type ρ_i of an object method is invariant since it appears covariantly in field get_{l_i} and contravariantly in put_{l_i}. Nonetheless, if an object O_1 (of type $(\mathbf{obj}\ \rho_1)$, with

Type System 4 Derived Type System for Objects

Let $\rho = [\, l_i : \rho_i^{\,1 \le i \le n}\,]$

$$\frac{\Gamma, x_i : (\mathbf{obj}\ \rho) \vdash P_i : \rho_i \qquad \Gamma, O : (\mathbf{obj}\ \rho) \vdash P : \tau}{\Gamma_{|O} \vdash \mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i) P_i^{\,1 \le i \le n} \right\} \mathbf{in}\ P : \tau} \qquad \frac{\Gamma \vdash O : (\mathbf{obj}\ \rho)}{\Gamma \vdash \mathbf{clone}(O) : (\mathbf{obj}\ \rho)}$$

$$\frac{\Gamma, x : (\mathbf{obj}\ \rho) \vdash P : \rho_j \qquad \Gamma \vdash Q : (\mathbf{obj}\ \rho)}{\Gamma \vdash Q \leftarrow l_j = \varsigma(x) P : (\mathbf{obj}\ \rho)} \qquad \frac{\Gamma \vdash P : (\mathbf{obj}\ \rho)}{\Gamma \vdash P \Leftarrow\!\!\!\!\!\!\Rightarrow l_j : \rho_j}$$

$\rho_1 = [\, l_i : {\rho_i}^{1 \le i \le n} \,]$) has less methods than another object O_2 (of type $(\mathbf{obj}\ \rho_2)$, with $\rho_2 = [\, l_i : {\rho_i}^{1 \le i \le n+m} \,]$), then $(\mathbf{obj}\ \rho_2) <: (\mathbf{obj}\ \rho_1)$ and O_2 can be used wherever O_1 can. We say that O_2 subsumes O_1.

Proposition 5 (Subsumption). $(\mathbf{obj}\ [\, l_1 : \rho_1, \ldots, l_{n+m} : \rho_{n+m} \,]) <: (\mathbf{obj}\ [\, l_1 : \rho_1, \ldots, l_n : \rho_n \,])$

The proof of this result follows from Proposition 1 and rule (sub rec) in Table 2. Subsumption, as polymorphism in **ML** or subtyping for records, is important to allow *reuse* of code, since it allows the definition of functions that behaves uniformly over inputs of different types. An example is the function: $(\lambda a)(a \Leftarrow\!= repaint)$, of type: $((\mathbf{obj}\ [\, repaint : \bot \,]) \to \bot)$, that can be applied to every objects with a method $repaint$ of type $\bot$.

4 Interpretation of Abadi and Cardelli Object Calculus

Abadi and Cardelli [1] have defined a functional calculus of primitive objects: ς, that formalizes aspects of object-oriented languages such as the notion of *self* (methods can refer to the object through self), *subsumption* (an object can emulate another object with fewer methods) or *method update*. In this paper, we study $\mathbf{Ob}_{1<:}$, a variant of ς with first-order type system and subtyping. Although ς has no notion of names or "identity", nor notion of concurrency, the constructs introduced in Table 3 are obviously inspired from Abadi-Cardelli's one. This is reflected by the simplicity of the interpretation of $\mathbf{Ob}_{1<:}$ given in this section.

Table 5 Objects, Reduction and Types in $\mathbf{Ob}_{1<:}$

x	variable
$\varsigma(x)b$	method with self parameter x and body b
$[\, l_i = \varsigma(x_i)b_i{}^{1 \le i \le n} \,]$	object with n methods labeled $l_1, \ldots, l_n$
$o \cdot l$	invocation of method l of object o
$o \cdot l \Leftarrow \varsigma(x)b$	update of method l of object o with method $\varsigma(x)b$

$$\frac{o \to_\varsigma v = [\, l_i = \varsigma(x_i)b_i{}^{1 \le i \le n} \,] \quad 1 \le j \le n}{o \cdot l_j \to_\varsigma b_j\{v/x_j\}}\ \text{(select)} \qquad \frac{o \to_\varsigma [\, l_i = \varsigma(x_i)b_i{}^{1 \le i \le n} \,] \quad 1 \le j \le n}{o \cdot l_j \Leftarrow \varsigma(x)b \to_\varsigma [\, l_j = \varsigma(x)b,\ l_i = \varsigma(x_i)b_i{}^{i \ne j} \,]}\ \text{(update)}$$

Let A be $[\, l_i : {B_i}^{1 \le i \le n} \,]$.

$$\frac{}{E, x : A \vdash x : A}\ \text{(axiom)} \qquad \frac{E, x_i : A \vdash b_i : B_i \quad \forall i\ 1 \le i \le n}{E \vdash [\, l_i = \varsigma(x_i)b_i{}^{1 \le i \le n} \,] : A}\ \text{(object)}$$

$$\frac{E \vdash a : A \quad 1 \le j \le n}{E \vdash a \cdot l_j : B_j}\ \text{(select)} \qquad \frac{E \vdash a : A \quad E, x : A \vdash b : B_j \quad 1 \le j \le n}{E \vdash a \cdot l \Leftarrow \varsigma(x)b : A}\ \text{(update)}$$

the relation of subtyping is such that $[\, l_i : {B_i}^{1 \le i \le n+m} \,] <: [\, l_i : {B_i}^{1 \le i \le n} \,]$.

$$\frac{E \vdash a : A \quad A <: B}{E \vdash a : B}\ \text{(subsumption)}$$

The syntax and semantics of $\mathbf{Ob}_{1<:}$ is given in Table 5. There is a unique binder in this calculus, namely ς, that binds occurrences of x in a method declaration: $\varsigma(x)b$. We consider

the weak reduction relation (such that reduction under binders is forbidden). In particular, terms of the form $[\, l_i = \varsigma(x_i)b_i{}^{1\le i\le n}\,]$ are values. We assume also that $o \to_\varsigma o'$ implies $\mathrm{E}\,[o] \to_\varsigma \mathrm{E}\,[o']$, where $\mathrm{E}\,[]$ are ς's evaluation contexts: $[.]$, $(\mathrm{E}\,[]\cdot l_i)$ and $(\mathrm{E}\,[]\cdot l_i \Leftarrow \varsigma(x)b)$.

In Table 6, we give a translation of $\mathbf{Ob}_{1<:}$ into our calculus of cloneable objects derived from $\boldsymbol{\pi}^\star$. With this interpretation, the operation of substituting a variable with a term is re-

Table 6 Interpretation of $\mathbf{Ob}_{1<:}$

$$[\![x]\!] \;=\; \mathbf{clone}(x) \qquad [\![o\cdot l]\!] \;=\; [\![o]\!] \Leftarrow\!\!\!\!\Rightarrow l \qquad [\![o\cdot l \Leftarrow \varsigma(x)b]\!] \;=\; ([\![o]\!] \leftarrow l = \varsigma(x)[\![b]\!])$$

$$[\![\,[\, l_i = \varsigma(x_i)b_i{}^{1\le i\le n}\,]\,]\!] \;=\; \mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i)[\![b_i]\!]^{1\le i\le n} \right\} \mathbf{in}\; O$$

$$[\![\,[\, l_i : B_i{}^{1\le i\le n}\,]\,]\!] = \; (\mathbf{obj}\; [\, l_i : [\![B_i]\!]^{1\le i\le n}\,]) \qquad [\![\emptyset]\!] = \; \emptyset \qquad [\![E, x : A]\!] = \; [\![E]\!],\; x : [\![A]\!]$$

placed by substituting a variable with the name of a resource where the term can be fetched. This is standard when you code a "term passing" calculus in a name passing calculus, as in the encoding of the λ-calculus in $\boldsymbol{\pi}$ [21]. The following theorem states that there is an operational correspondence between $(\mathbf{Ob}_{1<:}, \to_\varsigma)$ and $(\boldsymbol{\pi}^\star, \to)$ and that our interpretation of types is correct.

Theorem 6 ($\boldsymbol{\pi}^\star$ simulates $\mathbf{Ob}_{1<:}$). *Our encoding of* $\mathbf{Ob}_{1<:}$ *is* complete*:* $o \to_\varsigma o' \;\Rightarrow\; [\![o]\!] \to^* \approx [\![o']\!]$*, and* sound*:* $[\![o]\!] \to P \Rightarrow o \to_\varsigma o'$ *with* $[\![o]\!] \to^* \approx [\![o']\!]$*. Moreover the interpretation of types is correct. That is* $A <: B$ *iff* $[\![A]\!] <: [\![B]\!]$*, and* $E \vdash_{\mathbf{Ob}_{1<:}} a : A$ *implies* $[\![E]\!] \vdash_{\boldsymbol{\pi}^\star} [\![a]\!] : [\![A]\!]$.

The proof of Theorem 6 uses an intermediate result to prove that $\mathbf{obj}\, O = \{L\}\mathbf{in}\; [\![b]\!]\{O/x\}$ is operationally equivalent to $[\![b\{v/x\}]\!]$ whenever $(\mathbf{obj}\; O = \{L\}\; \mathbf{in}\; O)$ is the translation of v:

Lemma 7 ("object-replication" theorem). *Let* $\mathrm{M}\,[]$ *be a "multi-hole" context. If* O *does not appear in* $\mathrm{M}\,[]$*, then:* $\mathbf{obj}\, O = \{L\}\,\mathbf{in}\,\mathrm{M}\,[\mathbf{clone}(O)] \;\approx\; \mathrm{M}\,[\mathbf{obj}\, O = \{L\}\,\mathbf{in}\, O]$*. As a corollary, if* O *does not appear in* P *and* Q*, except possibly in a* $\mathbf{clone}(O)$ *statement, then:*

$$\begin{array}{ll} (1)\; \mathbf{obj}\, O = \{L\}\,\mathbf{in}\,(P \mid Q) & \approx\; (\mathbf{obj}\, O = \{L\}\,\mathbf{in}\, P) \mid (\mathbf{obj}\, O = \{L\}\,\mathbf{in}\, Q) \\ (2)\; \mathbf{obj}\, O = \{L\}\,\mathbf{in}\,\mathbf{clone}(O) & \approx\; \mathbf{obj}\, O = \{L\}\,\mathbf{in}\, O \end{array}$$

Our interpretation can be easily extended to an imperative variant of $\mathbf{Ob}_{1<:}$ by defining $[\![\mathbf{clone}(o)]\!] =_{\mathrm{def}} \mathbf{clone}([\![o]\!])$ and $[\![\mathbf{let}\, x = o\,\mathbf{in}\, b]\!] =_{\mathrm{def}} \mathbf{def}\; x = [\![o]\!]\,\mathbf{in}\,[\![b]\!]$ (we obtain the imperative ς-calculus of [1], but with a "call-by-name" let-operator). It is interesting to note that some equivalences true in the imperative calculus [15] are preserved by the translation.

Lemma 8 (Some Equivalences on $[\![\mathbf{Ob}_{1<:}]\!]$). *If* v *is the object* $[\, l_i = \varsigma(x_i)b_i{}^{1\le i\le n}\,]$*, then* $[\![\mathbf{clone}(v)]\!] \approx [\![v]\!]$ *and:*

$$[\![v\cdot l_j]\!] \approx \mathbf{def}\; x_j = [\![v]\!]\,\mathbf{in}\,[\![b_j]\!] \qquad [\![(v\cdot l_j \Leftarrow \varsigma(x)b)\cdot l_j]\!] \approx \mathbf{def}\; x_j = [\![v\cdot l_j \Leftarrow \varsigma(x)b]\!]\,\mathbf{in}\,[\![b]\!]$$

5 Conclusions and Future Work

We have presented a concurrent calculus of imperative objects derived from the blue calculus and we have used it to give an encoding of $\mathbf{Ob}_{1<:}$, a typed sequential object calculus, that preserves subtyping. Process calculi have been extensively used for reasoning on the foundation of object-oriented computation, but our aim was different here. Instead of focusing on the definition of a formal semantics for an existing programming languages, we have rather tried to prove that $\pi^\star$ is a suitable basis for the design of a higher-order concurrent programming language with object-oriented features. This goal was achieved by interpreting objects as a special kind of processes. In this respect, we follow Pierce's and Turner's works on the asynchronous π calculus, that was motivated by the design of the PICT [25] programming language.

The encoding of Abadi and Cardelli typed object calculus was a successful test for the expressiveness of $\pi^\star$, since we give a simple and direct (i.e. without use of a continuation passing style) typed interpretation of $\mathbf{Ob}_{1<:}$ in a first order process-calculus. It is interesting to note that our interpretation of types is similar to the one given in [26]. This lead us to the conjecture that our encoding can be extended to $\mathbf{Ob}_{1<:\mu}$, a variant of $\mathbf{Ob}_{1<:}$ with recursive types.

Development of the work presented here are being conducted at the moment. For example, using result obtained on $\pi^\star$ [10], we have defined a polymorphic type system "a la $\mathbf{ML}$" for our object calculus. There is also ongoing work aiming at adding a notion of location to $\pi^\star$ [11]. It would be interesting then to give an interpretation of distributed object-oriented languages, such as OBLIQ [9], that lacks a formal definition and techniques to reason about program equivalences, in a distributed blue calculus. Another application is the modeling of Object Request Brokers, like CORBA. Indeed, record types are reminiscent of the interface description language used in ORBs [22]. Thus, we believe that $\pi^\star$ is a calculus suitable for the verification of distributed objects applications built using these tools. Finally, it will be interesting to study the equivalence obtained on the ς-calculus using the operational equivalence, $\approx$, defined in Sect. 2.4 and the encodings of ς terms in $\pi^\star$. Some examples were given in Lemma 8.

Acknowledgments

I want to thank Gérard Boudol for his useful comments. Not only he is the originator of the blue calculus, but he is also the source of most of the ideas contained in this paper.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 2(125):78–102, March 1996.
3. Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Proceedings POPL'96*, pages 396–409, 1996.
4. H. P. Barendregt. *The Lambda Calculus, Its syntax and Semantics*. North Holland, 1981.
5. G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, 1992.
6. Gérard Boudol. The π-calculus in direct style. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, Paris, France, 15–17 January 1997.

7. Gérard Boudol. Typing the use of resources in a concurrent calculus. In *ASIAN'97, the Asian Computing Science Conference*, Lecture Notes in Computer Science, Kathmandu, December 1997.
8. L. Cardelli and J. C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991.
9. Luca Cardelli. A language with distributed scope. *Computer Systems*, 8:27–59, 1995.
10. Silvano Dal-Zilio. Implicit polymorphic type system for the blue calculus. Technical Report 3244, INRIA, September 1997.
11. Silvano Dal-Zilio. Quiet and bouncing objects: Two migration abstractions in a simple distributed blue calculus. In *1st International Workshop on Semantics of Objects as Processes*. BRICS Notes Series, July 1998.
12. P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *7th International Conference on Concurrency Theory CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1996.
13. Kathleen Fisher and John C. Mitchell. A delegation-based object calculus with subtyping. In *proceedings of FCT'95*, volume 965 of *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, 1995.
14. Andrew Gordon and D. Hankin. A concurrent object calculus. (draft), 1998.
15. Andrew Gordon, D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proceedings of FST & TCS'97*. Lecture Notes in Computer Science, December 1997.
16. C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.
17. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
18. Cliff B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.
19. Josva Kleist and Davide Sangiorgi. Imperative objects and mobile processes, 1998. (to appear in Proceedings of PROCOMET'98).
20. R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.
21. Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.
22. E. Najm and J.B. Stefani. A formal semantics for the ODP computational model. Technical Report PAA/3527, CNET, May 1993.
23. Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, INRIA, May 1991.
24. Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report 3000, INRIA, 1996.
25. David N. Turner. *The polymorphic pi-calculus: theory and implementation*. PhD thesis, University of Edinburgh, 1995.
26. Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. to appear in proceedings of LICS'98.
27. D. Walker. π-calculus semantics of object-oriented programming languages. In *Proceedings TACS'91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991. Proc. TACS'91.
28. Mitchell Wand. Complete type inference for simple objects. In *2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in LICS 1988.