Subtyping and Matching for Mobile Objects *

Michele Bugliesi[1], Giuseppe Castagna[2], and Silvia Crafa[1,2]

[1] Dipartimento di Informatica
Univ. "Ca' Foscari", Venezia, Italy

[2] Département d'Informatique
École Normale Supérieure, Paris, France

Abstract. In [BCC00], we presented a general framework for extending calculi of mobile agents with object-oriented features, and we studied a typed instance of that model based on Cardelli and Gordon's Mobile Ambients. Here, we refine our earlier work and define a new calculus which is based on *Remote Procedure Call* as the underlying protocol for method invocation, and on a different typing technique for method bodies. The new type system is equipped with a subtyping and a matching relation: the combination of matching with subtyping provides new insight into the relationship between ambient opening in the new calculus and method overriding in object-oriented calculi.

1 Introduction

Calculi of mobile agents are receiving increasing interest in the programming language community as advances in computer communications and hardware enhance the development of large-scale distributed programming. *Agents* are effective entities that perform computation and interact with other agents: the term "mobile" implies that agents are bound to *locations* and that this binding may vary over time; agent interaction, in turn, is achieved using resources such as communication channels.

Independently of the new trends in communication technology, object-oriented programming has established itself as the de-facto standard for a principled design of complex software systems.

Drawing on our earlier work [BC00,BCC00], in this paper we study a formal calculus that integrates object-oriented constructs into calculi of mobile agents. The resulting calculus provides foundations for a computation model for distributed applications, where conventional client-server technology —based on remote exchange of messages between static sites— and mobile agents coexist in a uniform way.

The model results from extending the structure of *named* agents in the style of Mobile Ambients [CG98] with method definitions and primitive constructs for *self* denotation and message passing. The extension has interesting payoffs, as it leads to a principled approach to structuring agents: specifically, introducing methods and message passing as primitive, rather than encoding them on top of the underlying calculus of agents leads to a rich and precise notion of agent interface and type. Furthermore, it opens the way to reusing the advances in type system of object-oriented programming and static analysis.

* Work partially supported by MURST Project 9901403824_003, by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet", and by Galileo Action n. 02841UD

With respect to our earlier work [BC00,BCC00] this paper brings two main contributions to the calculus. For the operational semantics, we study a new model of message passing and method invocation based on Remote Procedure Call (RPC)[1]. For the type system, we discuss a non-trivial blend of matching and subtyping relations. Method invocation based on RPC fits nicely the design of a typed distributed calculus as it allows method bodies to be type-checked locally, in the object where they are defined, independently of the caller. As a consequence, the choice of RPC as the underlying semantics of method invocation yields a notion of interface-type for our mobile objects that is substantially simpler and more tractable than the corresponding notion defined in [BCC00]. The combination of subtyping and matching, in turn, conveys new insight into the relationship between method overriding in object-oriented calculi and the open capability in our mobile objects. As we show, matching is necessary in the type system to ensure type soundness for object opening in the presence of subtping[2].

Plan of the paper In Section 2 we introduce the calculus of mobile objects, named MA^{++} , based on the calculus of Mobile Ambients by [Car99,CG98]. Section 3 illustrates the expressive power of the calculus with several, diversified, examples. In Section 4 we study the type theory of our calculus, and state relevant properties. Related work is discussed in Section 5. Final remarks in Section 6 conclude our presentation with a discussion on current and future work.

2 MA^{++}

The syntax of the calculus is essentially the same as that originally defined in [BCC00], and results from generalizing the structure of ambients to include method definitions, or *interfaces*, as in $a[\![\mathrm{I}\,;\,P]\!]$, where P is a process and I is a list of method definitions, defined by the following productions:

$$\begin{array}{llcll}
\text{Processes} & P & ::= & \mathbf{0} & \text{inactivity} \\
 & & | & P \mid P & \text{parallel composition} \\
 & & | & a[\![\mathrm{I}\,;\,P]\!] & \text{ambient} \\
 & & | & (\boldsymbol{\nu} x)P & \text{restriction} \\
 & & | & M.P & \text{action} \\
 & & & & \\
\text{Interfaces} & \mathrm{I} & ::= & \ell(\boldsymbol{x}) \triangleright \varsigma(z)P & \text{method} \\
 & & | & \mathrm{I} :: \mathrm{J} & \text{sequence} \\
 & & | & \varepsilon & \text{empty interface} \\
 & & & & \\
\text{Patterns} & \boldsymbol{x} & ::= & x & \text{variable} \\
 & & | & (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) & \text{tuple } (n \geqslant 1)
\end{array}$$

[1] RPC is often referred to as *Remote Method Invocation* (RMI) in this context.

[2] The new version of the type system also rectifies a flaw of the type system we presented in [BCC00].

The syntax of processes is a generalization of the combinatorial kernel of the Ambient Calculus: $\mathbf{0}$ denotes the inactive process, $P \mid Q$ the parallel composition of two processes P and Q, $a[\![\, \mathrm{I}\,;\, P\,]\!]$ denotes the object named a with interface I and enclosed process P, $(\boldsymbol{\nu} x)P$ restricts the name x to P, and finally $M.P$ performs the action described by the term M and then continues as P.

Interfaces are lists of labels with associated processes: the syntactic form $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$ denotes a method labeled ℓ whose associated body is the process P where the ς-bounded variable z is the *self* parameter distinctive of object calculi, representing the method's host object. Finally, the pattern $\boldsymbol{x}$ is the tuple of input parameters for P.

$$
\begin{array}{llcll}
\text{Terms} & M, N & ::= & a, b, \ldots, x, y \ldots & \text{name/variable} \\
 & & \mid & (M_1, \ldots, M_n) & \text{tuple } (n \geqslant 0) \\
 & & \mid & M.M & \text{path} \\
 & & \mid & \varepsilon & \text{empty path} \\
 & & \mid & \mathsf{in}\ a & \text{enter } a \\
 & & \mid & \mathsf{out}\ a & \text{exit } a \\
 & & \mid & \mathsf{open}\ a & \text{open } a \\
 & & \mid & a\ \mathsf{send}\ \ell\langle M \rangle & \text{remote invocation}
\end{array}
$$

Terms include the capabilities distinctive of Mobile Ambients. In addition, our ambients are equipped with a capability for remote method invocation: the expression $a\ \mathsf{send}\ \ell\langle M \rangle$ invokes the method labeled ℓ residing on the object denoted by a with arguments M.

In the following we let $P, Q, R, \ldots$ range over processes, I, J over (possibly empty) interfaces, and use lower case letters to denote generic names, reserving $a, b, \ldots$ for ambient names, and $x, y, \ldots$ for parameters, whenever possible. Method names, denoted ℓ, range over a disjoint alphabet and have a different status: they are fixed labels that may not be restricted, abstracted upon, nor passed as values (they are similar to field labels in record-based calculi). We omit trailing or isolated $\mathbf{0}$ processes and empty interfaces, using M, $a[\![\, \mathrm{I}\,]\!]$, $a[\![\, P\,]\!]$, and $a[\![\ ]\!]$ as shorthands for, respectively, $M.\mathbf{0}$, $a[\![\, \mathrm{I}\,;\, \mathbf{0}\,]\!]$, $a[\![\, \varnothing\,;\, P\,]\!]$, and $a[\![\, \varnothing\,;\, \mathbf{0}\,]\!]$. Throughout, we use the terms "ambient" and "object" interchangeably.

2.1 Operational Semantics

We define the operational semantics of the calculus by means of a structural congruence and a reduction relation. As usual, the former is used to rearrange a term in order to apply the latter.

Structural Congruence Structural congruence for processes is defined in terms of an auxiliary equivalence relation $\equiv_{\mathrm{I}}$ over interfaces, given in Figure 1. This relation allows method definitions be reordered without affecting the behavior of the enclosing object: reordering of methods, in turn, is used to define the reduction of method invocation.

Definitions for methods with different name and/or arity may freely be permuted (Eq Meth Comm); instead, if the same method has multiple definitions,

(Eq Meth Assoc)	$(\mathrm{I} :: \mathrm{J}) :: \mathrm{L}$	$\equiv_{\mathrm{I}} \mathrm{I} :: (\mathrm{J} :: \mathrm{L})$	
(Eq Meth Comm)	$\mathrm{I} :: m(\boldsymbol{x}_m) \triangleright P ; \ell(\boldsymbol{y}_\ell) \triangleright Q$	$\equiv_{\mathrm{I}} \mathrm{I} :: \ell(\boldsymbol{y}_\ell) \triangleright Q :: m(\boldsymbol{x}_m) \triangleright P$	$\ell \neq m$
(Eq Meth Over)	$\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright P :: \ell(\boldsymbol{x}) \triangleright Q :: \mathrm{I}$	$\equiv_{\mathrm{I}} \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright Q$	

Fig. 1. Equivalence for Methods

then the right-most definition overrides the remaining ones (Eq Meth Over). Similar notions of equivalence can be found in the literature on objects: in fact, our definition is directly inspired by the bookkeeping relation introduced in [FHM94].

Structural congruence of processes is defined as the smallest congruence that forms a commutative monoid with product $\mid$ and unit $\mathbf{0}$, and is closed under the rules in Figure 2, where the set *fn* of *free names* is defined by a standard extension of the definition in [Car99].

(Struct Res Dead)	$(\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0}$	
(Struct Res Res)	$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$	$x \neq y$
(Struct Res Par)	$(\boldsymbol{\nu} x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu} x)Q$	$x \notin fn(P)$
(Struct Res Amb)	$(\boldsymbol{\nu} p)a[\mathrm{I}\,;\, P] \equiv a[\mathrm{I}\,;\,(\boldsymbol{\nu} p)P]$	$p \notin fn(\mathrm{I}) \cup \{a\}$
(Struct Path Assoc)	$(M.M').P \equiv M.M'.P$	
(Struct Empty Path)	$\varepsilon.P \equiv P$	
(Struct Cong Amb Meth)	$\mathrm{I} \equiv_{\mathrm{I}} \mathrm{J} \Rightarrow a[\mathrm{I}\,;\, P] \equiv a[\mathrm{J}\,;\, P]$	

Fig. 2. Structural Congruence for Processes

The first block of clauses are the rules of the π-calculus. The rule (Struct Path Assoc) is a structural equivalence rule for the Ambient Calculus, while the rule (Struct Res Amb) modifies the rule for ambients in the Ambient calculus to account for the presence of methods. Rule (Struct Cong Amb Meth) establishes ambient equivalence up to reordering of method suites. In addition, we identify processes up to renaming of bound names: $(\boldsymbol{\nu} p)P = (\boldsymbol{\nu} q)P\{p := q\}$ if $q \notin fn(P)$.

Reduction Relation The reduction semantics of the calculus is given by the context rules in Figure 3, plus the notions of reduction collected in Figure 4, that we comment below.

$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \Rightarrow P' \rightarrow Q'$	$P \rightarrow Q \Rightarrow a[\mathrm{I}\,;\, P] \rightarrow a[\mathrm{I}\,;\, Q]$
$P \rightarrow Q \Rightarrow (\boldsymbol{\nu} x)P \rightarrow (\boldsymbol{\nu} x)Q$	$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$

Fig. 3. Structural Rules for Reduction

$$
\begin{array}{ll}
(\mathit{in}) & b[\mathrm{I};\ \mathsf{in}\ \ a.P \mid Q] \mid a[\mathrm{J}\,;R] \ \rightarrow\ a[\mathrm{I}\,;R \mid b[\mathrm{J}\,;P \mid Q]] \\
(\mathit{out}) & a[\mathrm{I}\,; b[\mathrm{J}\,; \mathsf{out}\ \ a.P \mid Q] \mid R] \ \rightarrow\ b[\mathrm{J}\,;P \mid Q] \mid a[\mathrm{I}\,;R] \\
(\mathit{open}) & \mathsf{open}\ \ a.P \mid a[Q] \ \rightarrow\ P \mid Q \\
(\mathit{update}) & b[\mathrm{I}\,; \mathsf{open}\ \ a.P \mid a[\mathrm{J}\,;Q] \mid R] \ \rightarrow\ b[\mathrm{I} :: \mathrm{J}\,; P \mid Q \mid R] \qquad \text{for } \mathrm{J} \neq \varepsilon \\
(\mathit{send}) & b[\mathrm{I}\,; a\ \mathsf{send}\ \ell\langle M\rangle.P \mid Q] \mid a[\mathrm{J} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,; S] \\
 & \qquad \rightarrow\ b[\mathrm{I}\,;P \mid Q] \mid a[\mathrm{J} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,; R\{z,\boldsymbol{x} := a, M\} \mid S]
\end{array}
$$

Fig. 4. MA^{++} reduction rules

The first three rules are exactly the same as the corresponding rules for Mobile Ambients. Rule (*update*) is a direct generalization of the open rule that handles the case when the opened ambient contains a non-empty set of method definitions. If a is one such ambient, $\mathsf{open}\ a$ may only be reduced within an enclosing ambient: as a result of reduction, the process local to a is unleashed within the opening ambient, and the interfaces of the opening and the opened ambients are merged as shown by the definition of the rule. The rule (*send*) handles the new syntactic construct for method invocation, implementing the RPC model. The notation $R\{z,\boldsymbol{x} := a, M\}$ indicates the simultaneous substitution in R of a for z and of M for $\boldsymbol{x}$. Informally, the result of the ambient b sending message ℓ to its sibling a, with argument M, is the activation of the process associated with ℓ on the receiver a, with M substituted for the input pattern $\boldsymbol{x}$ and the *self* parameter dynamically bound to the name of the receiver.

3 Expressive power

We discuss a number of constructs that can be expressed in our calculus, including constructs for method overriding distinctive of object calculi, various forms of process communication, as well as different primitives of method invocation. Some of these examples have been already presented in our earlier work [BCC00] where, however, they were defined in terms of a different semantics for method invocation based on *Code On Demand.* Throughout this section, we use the terms "protocol" and "encoding" as synonyms: technically, this is an abuse of terminology, as we don't claim the protocols to really be encodings, i.e. interference-free simulations of the constructs in question.

3.1 Parent-child and Local communications

As a first example, we look at alternative models for method invocation. Having having chosen RPC as our primitive semantics, we now discuss other models, such as those described in Figure 5, for sending messages from an ambient to its parent or children, or to its own methods.

$$
\begin{array}{lll}
(\mathit{downsend}) & a\ \mathsf{downsend}\ \ell\langle M\rangle.P \mid a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R \,]\!] & \\
 & \quad \rightarrow\ P \mid a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R \mid Q\{z := a, \boldsymbol{x} := M\} \,]\!] & \\
(\mathit{upsend}) & a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R \mid b[\![\, \mathrm{J}\,;\, a\ \mathsf{upsend}\ \ell\langle M\rangle.P \,]\!] \,]\!] & \\
 & \quad \rightarrow\ a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R \mid Q\{z := a, \boldsymbol{x} := M\} \mid b[\![\, \mathrm{J}\,;\, P \,]\!] \,]\!] & \\
(\mathit{local}) & a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, a\ \mathsf{local}\ \ell\langle M\rangle.P_1 \mid P_2 \,]\!] & \\
 & \quad \rightarrow\ a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, Q\{z, \boldsymbol{x} := M, a\} \mid P_1 \mid P_2 \,]\!] &
\end{array}
$$

Fig. 5. Other Constructs for Method Invocation

Parent-to-child invocation. The intended behavior for this form of method invocation can be obtained by defining the construct for downward method invocation as follows, where $p, q \notin \mathit{fn}(M) \cup \mathit{fn}(P)$):

$$a\ \mathsf{downsend}\ \ell\langle M\rangle.P \ \stackrel{\triangle}{=}\ (\boldsymbol{\nu} p, q)\,(p[\![\, a\ \mathsf{send}\ \ell\langle M\rangle.q[\![\, \mathsf{out}\ p \,]\!] \,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P)$$

Informally, we temporarily create a new ambient p that becomes a sibling of the receiver a on which it invokes the method; the ambient q is used for synchronization, to guarantee that the ambient p be destroyed only after the receiver has served the invocation. It is a routine check to verify that the desired effect of the invocation is achieved by a sequence of reduction steps. To ease the notation, we give the reduction steps in the simplified case of a method which does not have parameters and does not depend on *self* (neither of the two simplifications affects the protocol):

$$
\begin{array}{l}
a\ \mathsf{downsend}\ \ell.P \mid a[\![\, \ell \triangleright Q\,;\, R \,]\!] \\
\qquad \equiv\ (\boldsymbol{\nu} p, q)\Big(p[\![\, \underline{a\ \mathsf{send}\ \ell\langle M\rangle}.q[\![\, \mathsf{out}\ p \,]\!] \,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P\Big) \mid a[\![\, \ell \triangleright Q\,;\, R \,]\!] \\
\qquad \rightarrow\ (\boldsymbol{\nu} p, q)\,\big(p[\![\, q[\![\, \underline{\mathsf{out}\ p} \,]\!] \,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P\big) \mid a[\![\, \ell \triangleright Q\,;\, R \mid Q \,]\!] \\
\qquad \rightarrow\ (\boldsymbol{\nu} p, q)\,\big(p[\![\,]\!] \mid q[\![\,]\!] \mid \underline{\mathsf{open}\ q.\mathsf{open}\ p}.P\big) \mid a[\![\, \ell \triangleright Q\,;\, R \mid Q \,]\!] \\
\qquad \rightarrow^{*} P \mid a[\![\, \ell \triangleright Q\,;\, R \mid Q \,]\!]
\end{array}
$$

Local and Self Invocation. Local method invocation within an ambient a is encoded similarly to the previous case. Choosing $p, q \notin \mathit{fn}(M) \cup \mathit{fn}(P)$, one defines:

$$a\ \mathsf{local}\ \ell\langle M\rangle.P \stackrel{\triangle}{=} (\nu p, q)\,(p[\![\, \mathsf{out}\ a.a\ \mathsf{send}\ \ell\langle M\rangle.\mathsf{in}\ a.q[\![\, \mathsf{out}\ p \,]\!] \,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P)$$

Relying upon this definition, it is then easy to define self-invocation within method bodies. To exemplify, consider the following process:

$$a[\![\, \ell_1(x) \triangleright \varsigma(z)z\ \mathsf{local}\ \ell_2\langle x\rangle :: \ell_2(x) \triangleright P\,;\, R \,]\!]$$

Invoking the method ℓ_1 from outside the object a results in the execution of the process P in parallel with R within a.

Child-to-parent. We conclude our survey of alternative models of method invocation with a form of upward invocation, whereby an ambient invokes a method residing in the enclosing ambient. A first definition of the construct is simply

$$a \; \mathsf{upsend} \; \ell\langle M\rangle.P \;\triangleq\; \mathsf{out}\; a.a \; \mathsf{send}\; \ell\langle M\rangle.\mathsf{in}\; a$$

One problem with this definition is that it requires a move of the sender. As an alternative, one may envisage a different protocol that relies on an auxiliary ambient. Assume that the invocation occurs within an object b, and that b is directly enclosed into a:

$$\begin{aligned}&a \; \mathsf{upsend}\; \ell\langle M\rangle.P \triangleq\\ &\quad (\nu\, p,q)\,(p\boldsymbol{[}\,\mathsf{out}\; b.\mathsf{out}\; a.a\; \mathsf{send}\; \ell\langle M\rangle.\mathsf{in}\; a.\mathsf{in}\; b.q\boldsymbol{[}\,\mathsf{out}\; p\,\boldsymbol{]}\,\boldsymbol{]} \mid \mathsf{open}\; q.\mathsf{open}\; p.P)\end{aligned}$$

The definition is easily understood by simply looking at the chain of capabilities inside the ambient p. First, the ambient p exits its parent ambient b, then exits the ambient a (that contains the method to be invoked), then performs the message send and is finally destroyed after having opened the locking ambient q. It should be noted that a formal specification of the protocol requires that the definition be given parametrically with respect to the enclosing ambient (b in the definition given above).

3.2 Replication

The behavior of replication in concurrent calculi is typically defined by a structural equivalence rule establishing that $!P \equiv !P \mid P$. In our calculus, we can provide a similar construct by relying upon the implicit form of recursion underlying the reduction of method invocation. Let be $p, q \notin \mathit{fn}(P)$:

$$\begin{aligned}!P \;\triangleq\; (\boldsymbol{\nu} p,q)\;(&p\; \mathsf{downsend}\; !\langle\rangle.\mathsf{open}\; q.P \mid\\ &\qquad p\boldsymbol{[}\,! \triangleright \varsigma(z)(q\boldsymbol{[}\,\mathsf{out}\; z.z\; \mathsf{downsend}\; !\langle\rangle.\mathsf{open}\; q.P\,\boldsymbol{]}\,)\,;\;\boldsymbol{]}\,)\end{aligned}$$

The reduction for the encoding of $!P$ is then the following:

$$\begin{aligned}!P &\triangleq (\boldsymbol{\nu} p,q)\,\Big(\underline{p\; \mathsf{downsend}\; !\langle\rangle}.\mathsf{open}\; q.P \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(q\boldsymbol{[}\cdots\boldsymbol{]}\,)\,;\;\boldsymbol{]}\,\Big)\\ &\rightarrow (\boldsymbol{\nu} p,q)\,\big(\mathsf{open}\; q.P \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(\ldots)\,;\; q\boldsymbol{[}\,\underline{\mathsf{out}\; p}.p\; \mathsf{downsend}\; !\langle\rangle.\mathsf{open}\; q.P\,\boldsymbol{]}\,\boldsymbol{]}\,\big)\\ &\rightarrow (\boldsymbol{\nu} p,q)\,\big(\underline{\mathsf{open}\; q}.P \mid q\boldsymbol{[}\,p\; \mathsf{downsend}\; !\langle\rangle.\mathsf{open}\; q.P\,\boldsymbol{]} \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(\ldots)\,;\;\boldsymbol{]}\,\big)\\ &\rightarrow (\boldsymbol{\nu} p,q)\,(P \mid p\; \mathsf{downsend}\; !\langle\rangle.\mathsf{open}\; q.P \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(\ldots)\,;\;\boldsymbol{]}\,)\\ &\equiv P \mid !P\end{aligned}$$

Notice that there is just one capability ready to be exercised at each reduction step. Furthermore, the process P is activated only after the opening of the ambient q, hence it does not interfere with the protocol.

3.3 Code on Demand

We continue our series of examples showing a protocol for method invocation based on Code on Demand (CoD). The behavior CoD can be described as follows: a client c invokes a method ℓ on a server s; the server activates the method and then sends it back to the client for the latter to execute it. Formally this correspond to the following reduction rule:

$$\begin{array}{r} c[\![\, \mathrm{J}\,;\, s\ \mathsf{send\_cod}\ell\langle M\rangle.R \mid S\,]\!] \mid s[\![\, \mathrm{I} :: \ell(x) \triangleright \varsigma(z)Q\,;\, P\,]\!] \quad \rightarrow \\ c[\![\, \mathrm{J}\,;\, Q\{z, x := s, M\} \mid R \mid S\,]\!] \mid s[\![\, \mathrm{I} :: \ell(x) \triangleright \varsigma(z)Q\,;\, P\,]\!] \end{array}$$

The intended behavior can be obtained by defining the sender and the receiver ambients as follows:

$$\mathit{server} \stackrel{\triangle}{=} s[\![\, \mathrm{I} :: \ell(u, v, x) \triangleright \varsigma(z)u[\![\, \mathsf{out}\ z.\mathsf{in}\ v.Q\,]\!]\,;\, P\,]\!]$$

$$\mathit{client} \stackrel{\triangle}{=} c[\![\, \mathrm{J}\,;\, (\boldsymbol{\nu} p)s\ \mathsf{send}\ \ell\langle p, c, M\rangle.\mathsf{open}\ p.R \mid S\,]\!]$$

The protocol relies on the agreement between the server and the client upon the name of the ambient that carries the activated process back to the client. This name is decided locally by the client which passes it as an argument of the call together with its own name. Invoking $\ell\langle p, c.M\rangle$ spawns a new process on the server that simply carries the ambient p out of the server and back into the client c: once inside c, the transport ambient p is opened thus unleashing the process Q to be executed on the client.

The protocol can be refined by having the client pass a "return path" rather than just its name. In that case, the client would be in the position to choose where to receive and execute the requested method (e.g. , in one of its subambients).

3.4 Updates

The standard notion of method override in formal object calculi [AC96,FHM94] can be rephrased in our calculus, as follows:

> given the ambient $a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\, Q\,]\!]$ replace the current definition P of ℓ by the new definition P' to form the ambient $a[\![\, \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P'\,;\, Q\,]\!]$.

Method updates in this form can be expressed in our calculus by means of a protocol that uses an "updater" ambient to carry the new method body inside the ambient to be updated. The updater enters the ambient a to be updated, and the latter has a controlling process that opens the updater thus allowing updates on its own methods. The protocol is defined precisely below in an asynchronous setting, with the update defined as a process term: a similar encoding can be defined for synchronous updates. Moreover, the definition only allows local updates, in that an ambient may only override methods contained in subambients (of course other kind of updates can be expressed similarly)

A method update is denoted by a update $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$, read "the ℓ method at a gets definition P ", and is defined as the following process:

$$a \ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \ \stackrel{\triangle}{=}\ \textsc{upd}\boldsymbol{[}\, \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ \mathsf{in}\ a\,\boldsymbol{]}$$

The ambient to be updated may now be defined as follows:

$$a^{\star}\boldsymbol{[}\,\mathrm{I}\,;\,P\,\boldsymbol{]}\ \stackrel{\triangle}{=}\ a\boldsymbol{[}\,\mathrm{I}\,;\,!(\mathsf{open}\ \textsc{upd})\mid P\,\boldsymbol{]}$$

Now, if we form the composition a update $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P' \mid a^{\star}\boldsymbol{[}\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\,Q\,\boldsymbol{]}$, the reduction for open enforces the expected behavior:

$$a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P' \mid a^{\star}\boldsymbol{[}\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\,Q\,\boldsymbol{]}\quad \boldsymbol{\rightarrow}^{*}\quad a^{\star}\boldsymbol{[}\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P'\,;\,Q\,\boldsymbol{]}$$

Multiple updates for the same method may occur in parallel, in which case their relative order is established nondeterministically. The protocol, as defined, relies on the assumption that the name UPD of the updater carrying the new method body is "well known". A more realistic assumption is that the ambient to be updated and the context agree on the name of the updater prior to start the protocol. This can be accomplished with a different definition of the ambient to be updated, one that assumes that such ambients come with an ad-hoc method that sets the appropriate conditions for the actual update to take place. The *upd* method below serves this purpose.

$$a^{\star}\boldsymbol{[}\,\mathrm{I}\,;\,P\,\boldsymbol{]}\ \stackrel{\triangle}{=}\ a\boldsymbol{[}\,\mathrm{I} :: \mathit{upd}(u) \triangleright \varsigma(z)\mathsf{open}\ u\,;\,P\,\boldsymbol{]}$$

Now, the protocol comprises two steps. First the ambient to be updated receives the name of the updater, and only then does the update take place:

$$a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\ \stackrel{\triangle}{=}\ (\boldsymbol{\nu} p)\,(a\ \mathsf{downsend}\ \mathit{upd}\langle p\rangle.p\boldsymbol{[}\,\ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ \mathsf{in}\ a\,\boldsymbol{]}\,)$$

3.5 Encoding the π-calculus

As a final example, we define constructs for synchronous and asynchronous communication between processes (all processes, not just ambients) over named channels. Similar construts for channel-based communication are presented in [CG98], based on the more primitive form of local and *anonymous* communication defined for the Ambient Calculus. Here, instead, we rely on the ability, distinctive of our ambients, to exchange values between methods. We first give a construct for synchronous communication.

A named channel n is represented by an "updatable" ambient n, and three auxiliary ambients n^i, n^o and $\bar{n}$ used for synchronization. The ambient n defines a method `ch`: a process willing to read from n installs itself as the body of this method, whereas a process willing to write on n invokes `ch` passing along the argument of the communication.

$$\begin{aligned}
(ch\ n) &\stackrel{\triangle}{=} n^{\star}\boldsymbol{[}\,\mathtt{ch}(x) \triangleright \mathbf{0}\,\boldsymbol{]} \mid n^{i}\boldsymbol{[}\ \boldsymbol{]}\\
n!\langle y\rangle.Q &\stackrel{\triangle}{=} \mathsf{open}\ n^{o}.n\ \mathsf{downsend}\ \mathtt{ch}(y).\mathsf{open}\ \bar{n}.(n^{i}\boldsymbol{[}\ \boldsymbol{]} \mid Q)\\
n?(x).P &\stackrel{\triangle}{=} \mathsf{open}\ n^{i}.\ n\ \mathsf{update}\ \mathtt{ch}(x) \triangleright (\bar{n}\boldsymbol{[}\,\mathsf{out}\ n.P\,\boldsymbol{]}\,)\ .n^{o}\boldsymbol{[}\ \boldsymbol{]}
\end{aligned}$$

The steps of the communication protocol are as follows. A process $n?(x).P$ reading from n first grabs the input lock n^i provided by the channel, then installs itself as the body of the `ch` method in n, and finally releases the output lock n^o. Now the writing process can start its computation: after acquiring the lock n^o, it sends the message $\mathtt{ch}(y)$. The message activates the process $\bar{n}[\,\mathsf{out}\ n.P\{x := y\}\,]$ inside n. One further step brings the ambient $\bar{n}$ outside n where it is opened by the output process: this last step completes the synchronization phase of the protocol, and both processes may continue their computation. The output process releases a new input lock to reset the channel to its initial condition, and the protocol is completed.

Asynchronous communications are obtained directly from the protocol above, by a slight variation of the definition of $n!\langle A\rangle.Q$. We simply need a different way of composing Q with the context:

$$n!\langle y\rangle.Q \ \triangleq\ (\mathsf{open}\ n^o.n\ \mathsf{downsend}\ \mathtt{ch}(y).\mathsf{open}\ \bar{n}.(n^i[\ ]))\mid Q$$

Based on this technique, we can encode the synchronous (and similarly, the asynchronous) polyadic π-calculus in ways similar to what is done in [CG99]. Each name n in the π-calculus becomes a quadruple of names in our calculus: the name n of the ambient dedicated to the communication, the names n^i and n^o of the two locks, and the name $\bar{n}$ of the auxiliary ambient. Therefore, communication of a π-calculus name becomes the communication of a quadruple of ambient names.

$$\begin{array}{rcl}
\langle\!\langle (\boldsymbol{\nu} n)P \rangle\!\rangle & \triangleq & (\boldsymbol{\nu} n, \bar{n}, n^i, n^o)(n^i[\ ] \mid n^{\star}[\mathtt{ch}(x,\bar{x},x^i,x^o) \triangleright \mathbf{0}] \mid \langle\!\langle P\rangle\!\rangle) \quad \bar{n}, n^i, n^o \notin fn(\langle\!\langle P\rangle\!\rangle)\\
\langle\!\langle n!\langle y\rangle.Q \rangle\!\rangle & \triangleq & \mathsf{open}\ n^o.n\ \mathsf{downsend}\ \mathtt{ch}(y,\bar{y},y^i,y^o).\mathsf{open}\ \bar{n}.(n^i[\ ] \mid \langle\!\langle Q\rangle\!\rangle)\\
\langle\!\langle n?(x).P \rangle\!\rangle & \triangleq & \mathsf{open}\ n^i.n\ \mathsf{update}\ \mathtt{ch}(x,\bar{x},x^i,x^o) \triangleright (\bar{n}[\,\mathsf{out}\ n.\langle\!\langle P\rangle\!\rangle\,])\ .n^o[\ ]\\
\langle\!\langle P \mid Q \rangle\!\rangle & \triangleq & \langle\!\langle P\rangle\!\rangle \mid \langle\!\langle Q\rangle\!\rangle\\
\langle\!\langle !P \rangle\!\rangle & \triangleq & !\langle\!\langle P\rangle\!\rangle\\
\langle\!\langle \mathbf{0} \rangle\!\rangle & \triangleq & \mathbf{0}
\end{array}$$

Fig. 6. Encoding of the synchronous π-calculus

The initialization of the `ch` method in the ambient that represents the channel n could be safely omitted, without affecting the operational properties of the encoding. However, as given, the definition scales smoothly to the case of a typed encoding, preserving well-typing.

4 Types and Type Systems

The structure of ambient, capability and process types is similar to that of companion type systems for Mobile Ambients: their intended meaning, instead, is different.

$$
\begin{array}{lll}
\text{Signatures} & \Sigma ::= & (\ \ell_i(\mathscr{V}_i)\ )^{i\in I} \\
\text{Ambients} & \mathscr{A} ::= & \mathsf{Amb}[\Sigma] \\
\text{Capabilities} & \mathscr{C} ::= & \mathsf{Cap}[\Sigma] \\
\text{Processes} & \mathscr{P} ::= & \mathsf{Proc}[\Sigma] \\
\text{Values} & \mathscr{V} ::= & \mathscr{A} \mid \mathscr{C} \\
\text{Types} & \mathscr{T} ::= & X \mid \mathscr{A} \mid \mathscr{C} \mid \mathscr{P}
\end{array}
$$

Signatures convey information about the interface of an ambient, by listing the ambient's method names and their input types. The type $\mathsf{Amb}[\Sigma]$ is the type of ambients with methods declared in Σ, while the types $\mathsf{Cap}[\Sigma]$ and $\mathsf{Proc}[\Sigma]$ are the types of capabilities and processes, respectively, whose enclosing ambient (if any) has a signature containing at least the methods included in Σ.

The type $\mathscr{V}$ identifies the type of the expressions that may occur as arguments for method invocation, and defines them to be ambient names and capabilities. The complete syntax of types includes type variables, which are used in the typing rules for the typing of method bodies, as we explain shortly.

4.1 Subtyping and Matching

To enhance the flexibility of ambient typing and mobility, a subtype relationship is introduced over capability and process types, as defined by the two following core rules.

$$
\text{(SUB CAP)}\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \leq \mathsf{Cap}[\Sigma']}
\qquad
\text{(SUB PROC)}\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \leq \mathsf{Proc}[\Sigma']}
$$

Informally, the rules state that a capability (resp. process) type $\mathsf{Cap}[\Sigma]$ (resp. $\mathsf{Proc}[\Sigma]$) is a subtype of any capability (resp. process) type whose associated signature (set theoretically) contains Σ. The resulting relation of subtyping is reminiscent of the relation of subtyping in *width* distinctive of type systems for object calculi. Width subtyping is restricted to capability and process types, and does *not* extend to ambient types, as the extension would break type soundness in the presence of ambient opening. The reason is explained, intuitively, as follows: when opening an ambient, one needs *exact* knowledge of the contents of that ambient —specifically, of what exactly is the set its methods and their types— so as to ensure that the possible method overrides resulting from the opening be traced in the types.

As a result of capability and process subtyping, it is nevertheless possible, from within an ambient with interface Σ, to open any enclosed ambient with interface $\Sigma' \subseteq \Sigma$, where the inclusion may be strict. To account for this flexibility, we introduce a relation of *matching* [Bru94] over ambient types to complement the subtype relation over capability and process types. The relation of matching is defined by the following rule:

$$
\text{(MATCH AMB)}\ \frac{\Gamma \vdash \diamond \quad \Sigma' \subseteq \Sigma}{\Gamma \vdash \mathsf{Amb}[\Sigma] \mathrel{<\!\!\#} \mathsf{Amb}[\Sigma']}
$$

The complete definition of subtyping and matching includes standard rules for reflexivity and transitivity (not shown). Also, as customary, the subtyping relation is endowed in the type system via a subsumption rule, while matching is not.

A further remark is in order to explain the role of type variables in the syntax of types. As we noted, due to the presence of ambient opening, a method residing in ambient, say a, may be re-installed inside any ambient, say b, that opens a; furthermore, the (sub)typing rules provide guarantees that b has "more methods" than a. Now, in order for the original typing of the methods residing in a to be sound after the methods have been re-installed in b, one must ensure that the bodies of these methods be type-checked under appropriate assumptions for the type of *self*: specifically, this type should be so defined as to represent the type of all ambients where the methods may eventually be re-installed, via opening. This is accomplished in the type system by typing method bodies in type environments that assume the so-called *MyType* [Bru94] typing for the *self* variable, i.e. a match-bounded type variable $X \lessdot\!\!\# \mathscr{A}$, where $\mathscr{A}$ is the type of the ambient where the methods are initially installed.

Our relation of matching, and the technique of *MyType* typing of methods we just outlined are simplified versions of the corresponding relation and technique originally introduced in [Bru94]. The simplifications result from the syntax of types, and specifically from our ambient types being *simple*, i.e. not containing occurrences of type variables (neither free, nor bound). As a consequence, the type system does not support *MyType* method specialization [Bru94,FHM94], the OO-typing technique that allows method-types to be specialized when methods are inherited (or, in our context, when they are subsumed in an opening ambient). Instead, in our calculus a method body has always the same type (the one declared in Σ), independently of the dynamic binding of its *self* variable. This is not surprising, as our method bodies are processes with no return value, hence they are dealt with essentially as methods with return type `unit` in imperative object calculi.

4.2 Judgements and Typing Rules

The typed syntax of the calculus is described by the productions below:

$$\begin{array}{lll} \text{Interfaces} & \mathrm{I} & ::= \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \mid \mathrm{I} :: \mathrm{I} \mid \varepsilon \\ \text{Processes} & P & ::= \mathbf{0} \mid P|P \mid a[\![\mathrm{I};\, P]\!] \mid (\boldsymbol{\nu} x{:}\mathscr{A})P \mid M.P \\ \text{Expressions} & M & ::= x \mid (M_1, \ldots, M_n) \mid x \;\mathsf{send}\; \ell\langle M\rangle \mid \mathsf{in}\; x \mid \mathsf{out}\; x \mid \mathsf{open}\; x \mid \varepsilon \end{array}$$

The only type annotations in the syntax are those introduced by the restriction operator: the types for all the other variables are directly inferred from the existing annotations. Also note that we take method names to be fixed *labels* that may not be passed as values, nor restricted. The first restriction is justified by the fact that method names are part of the structure of ambient (capability and process) types; as a consequence, lifting this restriction would be possible but it would make our types (first-order) dependent types. Instead, lifting the second restriction is possible, and in fact not difficult, even though it complicates the format of the typing rules. For this reason we will disregard this issue in what follows.

Type environments are lists of term and type variable declarations, as defined by the following productions: $\Gamma ::= \varnothing \mid \Gamma, x : \mathscr{W} \mid \Gamma, X \lessdot\!\# \mathscr{A}$. The typing rules derive the following judgement forms, where we let $\mathscr{W}$ range over the set $\{X, \mathscr{A}, \mathscr{C}\}$ of extended value types:

$\Gamma \vdash M : \mathscr{W}$	M has type $\mathscr{W}$
$\Gamma \vdash X \lessdot\!\# \mathscr{A}$	X matches $\mathscr{A}$
$\Gamma \vdash P : \mathscr{P}$	P has type $\mathscr{P}$
$\Gamma \vdash \mathscr{T}$	well-formed type
$\Gamma \vdash \diamond$	well-formed type environment

The complete set of typing rules is presented in Appendix A, the most interesting are discussed below. We start with the rule for typing ambient opening.

$$\text{(OPEN)}\quad \frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma]}{\Gamma \vdash \mathsf{open}\ a : \mathsf{Cap}[\Sigma]}$$

As we noted earlier, opening an ambient requires precise knowledge of the type of the ambient being opened: this is expressed in the rule by fact that the type of the ambient a is an ambient type, not a type variable. Opening a is now legal under the condition that the signature of the opening ambient be equal to (in fact, contain, given the presence of subtyping) the signature of the ambient being opened. This condition is necessary for type soundness, as it guarantees that an ambient may only update existing methods of the opening ambient, preserving their original types.

$$\text{(MESSAGE)}\quad \frac{\Gamma \vdash a : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \lessdot\!\# \mathsf{Amb}[\ \ell(\mathscr{V}')\ ] \quad \Gamma \vdash M' : \mathscr{V}'}{\Gamma \vdash a\ \mathsf{send}\ \ell\langle M' \rangle : \mathsf{Cap}[\Sigma]}$$

The rule (MESSAGE) states that invoking method ℓ on an ambient a requires the type of a to match an ambient type containing the method ℓ. Note that the type of a may either be an ambient type matching (i.e. "longer" then) $\mathsf{Amb}[\ell(\mathscr{V}')]$, or else an unknown type (i.e. a type variable) occurring match-bounded in the context Γ. Since the body of the invoked method is activated on the receiver (rather than on the sender) no constraint is required on the type of the send capability. Of course, in order for the expression to type check, the message argument and the method parameters must have the same type[3].

$$\text{(AMB)}\quad (\Sigma = (\ \ell_i(\mathscr{V}_i)\ )^{i \in I})$$
$$\frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma] \quad \Gamma, Z \lessdot\!\# \mathsf{Amb}[\Sigma], z{:}Z, x_i{:}\mathscr{V}_i \vdash P_i : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash a[(\ell_i(x_i) \triangleright \varsigma(z)P_i)\ ^{i \in I}\ ;\ P] : \mathsf{Proc}[\Sigma']}$$

[3] In fact, since capability types can be subtyped, the type of the arguments can be subtypes of the type of the formal parameters.

The rule (Amb) for typing ambients is similar to the typing rule for objects in the calculus of extensible objects of [BB99]. Each method of the ambient is type-checked under the assumptions that (*i*) the self parameter has a type that matches the type of the enclosing ambient, (*ii*) method parameters have the declared type, and (*iii*) the type of each method body be consistent with the type of the enclosing ambient. As we noted earlier, the use of the match-bound type variable Z as the type of *self* ensures that methods local to ambient a are well-typed also within any other ambient that might eventually open a. On the other hand, the typing rule does not support *MyType* method specialization, as the types of method bodies are independent of the type of *self*.

Also note that the rule requires exact knowledge of the type of the ambient a: a structural rule allowing the name of the ambient to be typed with a match-bounded type variable would break type soundness, since we would not have a precise control of the openings of that ambient (see rule (open)). Finally, no constraint is imposed on the signature Σ', associated with the process type in the conclusion of the rule, as that signature is (a subset of) the signature of the ambient enclosing a (if any).

4.3 Subject Reduction and Type Soundness

We conclude the description of the basic type system with a result of subject reduction. The proof is rather standard, and only sketched due to lack of space.

Lemma 1 (Substitution).

1. *If* $\Gamma, x : \mathscr{W} \vdash P : \mathscr{P}$ *and* $\Gamma \vdash M : \mathscr{W}$*, then* $\Gamma \vdash P\{x := M\} : \mathscr{P}$.
2. *If* $\Gamma, Z \lessdot\!\# \mathscr{A}, z : Z \vdash P : \mathscr{P}$ *and* $\Gamma \vdash a : \mathscr{A}'$*,* $\Gamma \vdash \mathscr{A}' \lessdot\!\# \mathscr{A}$*,* *then* $\Gamma \vdash P\{z := a\} : \mathscr{P}$.

Proof. *By induction on the derivation of the first judgment in hypothesis.*

Lemma 2 (Subject Congruence).

1. *If* $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.
2. *If* $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $Q \equiv P$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.

Proof. *By simultaneous induction on the derivations of* $P \equiv Q$ *and* $Q \equiv P$.

Lemma 3 (Bounded Weakening).

1. *If* $\Gamma, x : \mathscr{W} \vdash P : \mathscr{P}$ *and* $\Gamma \vdash \mathscr{W}' \leq \mathscr{W}$ *then* $\Gamma, x : \mathscr{W}' \vdash P : \mathscr{P}$.
2. *If* $\Gamma, Z \lessdot\!\# \mathscr{A}, z{:}Z \vdash P : \mathscr{P}$ *and* $\Gamma \vdash \mathscr{A}' \lessdot\!\# \mathscr{A}$ *then* $\Gamma, Z \lessdot\!\# \mathscr{A}', z{:}Z \vdash P : \mathscr{P}$.

Proof. *By induction on the derivation of the first judgment in hypothesis.*

Theorem 1 (Subject Reduction).
If $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $P \rightarrow Q$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.

Proof. *By induction on the derivation of* $P \rightarrow Q$*, and a case analysis on the last applied rule.*

Besides being interesting as a meta-theoretical property of the type system, subject reduction may be used to derive a type safety theorem ensuring the absence of run-time (type) errors for well-typed programs. The errors we wish to statically detect are those of the kind "message not understood" distinctive of object calculi. With the current definition of the reduction relation such errors may not arise, as not-understood messages simply block: this is somewhat unrealistic, however, as the result of sending a message to an object (a server) which does not contain a corresponding method should be (and indeed is, in real systems) reported as an error.

To state and formalize type safety, we instrument the reduction relation with an additional error reduction, state as follows:

$$a[\![\mathrm{I} ; P \mid b \ \mathsf{send}\ \ell\langle M\rangle.Q]\!] \mid b[\![\mathrm{J} ; R]\!] \ \rightarrow\ a[\![\mathrm{I} ; P \mid \mathtt{ERR}]\!] \mid b[\![\mathrm{J} ; R]\!] \qquad (\ell \notin \mathrm{J})$$

where ERR is a distinguished process, with no type. The intuitive reading of the reduction is that a not-understood message causes a local error —for the sender of that message— rather than a global error for the entire system. The rule above is meaningful also in the presence of multiple ambients with equal name, as our type system (like those of [CG99,CGG99,LS00]) ensures that ambients with the same name have also the same type.

It is easy to verify that no system containing an occurrence of ERR can be typed in our type system. Type safety, i.e. absence of run-time errors may now be stated follows:

Theorem 2 (Soundness). *Let P be a well-typed MA^{++} process. Then, there exist no context* $\mathbf{C}[-]$ *such that* $P \rightarrow^* \mathbf{C}[\mathtt{ERR}]$.

5 Related work

In the literature on concurrent object-oriented programming, papers can be classified in two basic categories. The first category includes papers that provide semantics to objects by encoding them into process calculi. Examples of systematic translations of objects into the π-calculus can be found, for instance, in [Wal95,HK96,San98,KS98].

Papers in the second category propose formal calculi where primitive constructs for objects and for concurrent processes coexist. Within this class, one can further distinguish two complementary approaches. In the first, high-level object-oriented constructs are defined on top of name-passing process calculi [Vas94,PT95,FMLR00]. In the second, primitives for concurrency are built on top of imperative object calculi, in ways related to those we have discussed in this paper. Below we present a detailed discussion on papers closest to ours.

Gorgon and Hankin's concς-calculus [GH98]. The *concς*-calculus is a concurrent object calculus that results from Abadi and Cardelli's imperative object calculus by the addition of primitive constructs for parallel composition, restriction and synchronization via mutexes. Type systems for the calculus may be defined by sound extensions of existing type systems for the underlying object calculus to accommodate concurrency.

There are several similarities between *conc*ς and our calculus. In particular, the semantics of method invocation, based on self-substitution was directly inspired by [GH98]. As in our semantics, in [GH98] objects are explicitly named, and what gets substituted for the *self* variable is the name of the object rather then the object itself.

The fundamental difference between the work of [GH98] and ours is that *conc*ς does not address process mobility. In [GH98] distribution is completely disregarded, while in our framework objects may move through a hierarchy of nested locations, and communication (method invocation) often requires mobility. Moreover, due to the interplay between the dynamic nesting of ambients and the communication primitives, more method invocation styles can be modeled in our framework. A further difference is that the syntax of *conc*ς includes sequential composition of expressions that return results. This contrasts with the standard practice in process-based calculi [Vas94,PT95,Wal95,KS98], where the operation of returning a result is translated into sending a message on a result channel. Even though we did not explicitly address the problem of returning a result, it is easy to extend our framework by endowing agent interfaces not only with methods, but also with *fields* whose invocation returns an expression.

A distributed version of *conc*ς is studied in [Jef00], where the syntax of the calculus is enriched with a notion of *location*, and threads are allowed to migrate across locations. A basic difference with our approach is that in [Jef00] the author assumes a flat topology of locations, in which no explicit routing is required for mobility, and locations may *not* be created dynamically. Furthermore, in [Jef00] only a subset of objects (*serializable* objects) can be sent across the network, and only the so-called *located objects* can be accessed via remote threads.

The Ojeblik calculus [NHKM99]. Ojeblik is a concurrent object-based language built on top of Obliq [Car95], Cardelli's lexically scoped distributed programming language. In Ojeblik (and Obliq) object mobility is rendered by means of a migration mechanism that is accomplished by creating a copy of the object at the target site and then modifying the original (local) object such that it forwards future requests to the new (remote) object: The lexical scope rules of Obliq allow the aspects of distribution to safely be disregarded: object migration is then *correct* if the behavior of an object is transparent to whether the object has migrated or not.

Our approach is very different. As in Mobile Ambients, we assume that the process $a[\![\mathrm{I}\,;\, P]\!]$ is an abstraction for both an agent (client) and an object (server). This implies that in our framework mobile objects move without the burden of future obligations at the source location. A client agent willing to invoke a method of a server object, in turn, must approach the server in order to start the communication protocol. In addition, while the work on Ojeblik does not address typing issues, as we do for our calculus.

6 Current and Future Work

We have defined a core calculus for distributed and mobile objects on top of which several extensions can be defined. We conclude our presentation with a discussion on some of these extensions.

Co-capabilities à la Safe Ambients. In [LS00], Levi and Sangiorgi define a variant of Mobile Ambients in which the reduction relation requires actions (i.e. capabilities) to synchronize with corresponding co-actions. To exemplify, consider the ambients $a[\![\, \mathsf{in}\ b.P \,]\!] \mid b[\![\, Q \,]\!]$. In mobile ambients, the move of a into b is "one sided" as b simply undergoes the action. In Safe Ambients, instead, the move requires mutual agreement between a and b: in order for the move to take place, Q inside b must offer the co-capability $\mathsf{coin}\ b$ to signal that it is willing to be entered. Based on this synchronization mechanisms, Levi and Sangiorgi discuss a suite of type systems for on top of which they develop a rich algebraic theory for their Safe Ambients.

Co-capabilities can be included in our calculus with no fundamental difficulty. In particular, one can include a co-capability $\mathsf{listen}\ a$, the dual of the capability $a\ \mathsf{send}$, whose meaning is that the ambient a is ready to serve an invocation to one of its methods. For reasons of space, we do not describe the extension in detail. Nevertheless, it is instructive to point out one of the effects of the extension, showing how it allows us to derive a simple compositional encoding of the π-calculus.

$$\begin{aligned}
\langle\!\langle\, n?(\boldsymbol{x}).P \,\rangle\!\rangle &\triangleq (\boldsymbol{\nu} p)(n[\![\, ch(\boldsymbol{x}) \triangleright p[\![\, \mathsf{out}\ n.\mathsf{coopen}\ p.\langle\!\langle\, P \,\rangle\!\rangle \,]\!]\ ;\ \mathsf{listen}\ n.\mathsf{coout}\ n \,]\!] \mid \mathsf{open}\ p)\\
\langle\!\langle\, n!\langle \boldsymbol{x} \rangle \,\rangle\!\rangle &\triangleq n\ \mathsf{downsend}\ ch\langle \boldsymbol{x} \rangle\\
\langle\!\langle\, (\boldsymbol{\nu x})P \,\rangle\!\rangle &\triangleq (\boldsymbol{\nu x})\langle\!\langle\, P \,\rangle\!\rangle\\
\langle\!\langle\, P \mid Q \,\rangle\!\rangle &\triangleq \langle\!\langle\, P \,\rangle\!\rangle \mid \langle\!\langle\, Q \,\rangle\!\rangle\\
\langle\!\langle\, !P \,\rangle\!\rangle &\triangleq !\langle\!\langle\, P \,\rangle\!\rangle\\
\langle\!\langle\, \mathbf{0} \,\rangle\!\rangle &\triangleq \mathbf{0}\\
\langle\!\langle\, n \,\rangle\!\rangle &\triangleq n
\end{aligned}$$

Every input on a channel n generates a new ambient named n, waiting to synchronize with an output on n. Having received input, the transport ambient p carries (the encoding of) P out of n. Once outside n, p is dissolved and the continuation process P unleashed. Notice that the ambient n is left without capabilities after having let the transport p out. As such, after synchronization, n is unavailable for interactions with the context, and thus behaviorally equivalent to the null process (which can be garbage collected). Also, the encoding can be shown to be interference-free, as the use of co-capabilities allows the definition of an interference-free encoding of output construct of the π-calculus, based on downward method invocation.

Other Extensions. Further extensions to the core calculus include the addition of fields and refinements of the type system.

In object calculi, fields are often represented as parameter-less methods, that do not depend on self. This direct representation is not possible in our calculus, as invoking a method spawns a process rather than returning a value, as one would expect from selecting a field. Nevertheless, it is not difficult of explicitly include new syntax for fields, and extend the reduction relation so that selecting a field returns a term rather than triggering a process.

A different extension is to allow method names to be treated as ordinary names. This would allow one to restrict them, thus obtaining private methods, and to communicate them, thus obtaining dynamic messages. This is a straightforward modification in the untyped calculus but it is quite problematic in the typed case since the possibility of communicating method names would naturally give rise to dependent types.

These extensions, together with the study of type-driven security in the calculus are subject of our current and future work.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[BB99] V. Bono and M. Bugliesi. Matching for the Lambda Calculus of Objects. *Theoretical Computer Science*, 212(1/2):101–140, Feb. 1999.

[BC00] M. Bugliesi and G. Castagna. Mobile objects. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. Electronic Proceedings.

[BCC00] M. Bugliesi, G. Castagna, and S. Crafa. Typed mobile objects. In *Proceedings of CONCUR 2000 (11th. International Conference on Concurrency Theory)*, number 1877 in Lecture Notes in Computer Science, pages 504–520. Springer, 2000.

[Bru94] B. Bruce, K. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 1(4):127–206, 1994.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Car99] L. Cardelli. Abstractions for mobile computations. In *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science, pages 51–94. Springer, 1999.

[CG98] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL '98*. ACM Press, 1998.

[CG99] L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL '99*, pages 79–92. ACM Press, 1999.

[CGG99] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP '99*, number 1644 in Lecture Notes in Computer Science, pages 230–239. Springer, 1999.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FMLR00] Cédric Fournet, Luc Maranget, Cosimo Laneve, and Didier Rémy. Inheritance in the Join Calculus. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*. Springer, December 2000.

[GH98] A. Gordon and P. D Hankin. A concurrent object calculus: reduction and typing. *In Proceedings HLCL '98, Elsevier ENTC*, 1998. Also Technical Report 457, University of Cambridge Computer Laboratory, February 1999.

[HK96] H. Huttel and J. Kleist. Objects as mobile processes. Technical Report Research Series RS-96-38, BRICS, 1996. Presented at MFPS '96.

[Jef00] A. Jeffrey. A distributed object calculus. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. Electronic Proceedings.

[KS98] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *PROCOMET '98 (IFIP Working Conference on Programming Concepts and Methods)*. North-Holland, 1998.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL '00*, pages 352–364. ACM Press, 2000.

[NHKM99] U Nestmann, H. Huttel, J. Kleist, and M. Merro. Aliasing models for object migration. In *Proceedings of Euro-Par '99*, number 1685 in Lecture Notes in Computer Science, pages 1353–1368. Springer, 1999.

[PT95] B.C. Pierce and D.N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming, Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer, April 1995.

[San98] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998.

[Vas94] V.T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *ECOOP '94*, number 821 in Lecture Notes in Computer Science, pages 100–117. Springer, 1994.

[Wal95] D.J Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

A Typing rules

Context formation

$$\text{(ENV-EMPTY)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(ENV-}x\text{)}\ \frac{\Gamma \vdash \mathscr{W} \quad x \notin Dom(\Gamma)}{\Gamma, x : \mathscr{W} \vdash \diamond} \qquad \text{(ENV-}X\text{)}\ \frac{\Gamma \vdash \diamond \quad X \notin Dom(\Gamma)}{\Gamma, X {<\!\!\#}\, \mathscr{A} \vdash \diamond}$$

Type formation

$$\text{(TYPE X)}\ \frac{\Gamma, X {<\!\!\#}\, \mathscr{A}, \Gamma' \vdash \diamond}{\Gamma, X {<\!\!\#}\, \mathscr{A}, \Gamma' \vdash X} \qquad \text{(TYPE AMB)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[\Sigma]} \qquad \text{(TYPE CAP)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Cap}[\Sigma]} \qquad \text{(TYPE PROC)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Proc}[\Sigma]}$$

Matching : Reflexivity, Transitivity and the following

$$\text{(MATCH X)}\ \frac{\Gamma, X {<\!\!\#}\, \mathscr{A}, \Gamma' \vdash \diamond}{\Gamma, X {<\!\!\#}\, \mathscr{A}, \Gamma' \vdash X {<\!\!\#}\, \mathscr{A}} \qquad \text{(MATCH AMB)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i \in 1..n+k}] {<\!\!\#}\, \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i \in 1..n}]}$$

Subtyping and subsumption : Reflexivity, Transitivity and the following

$$\text{(SUB CAP)}\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \leq \mathsf{Cap}[\Sigma']} \qquad \text{(SUB PROC)}\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \leq \mathsf{Proc}[\Sigma']} \qquad \text{(SUBSUMPTION)}\ \frac{\Gamma \vdash A : \mathscr{T} \quad \mathscr{T} \leq \mathscr{T}'}{\Gamma \vdash A : \mathscr{T}'}$$

Expressions

$$\text{(NAME/VAR)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)} \qquad (\varepsilon)\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \varepsilon : \mathsf{Cap}[\Sigma]} \qquad \text{(PATH)}\ \frac{\Gamma \vdash M_1 : \mathsf{Cap}[\Sigma] \quad \Gamma \vdash M_2 : \mathsf{Cap}[\Sigma]}{\Gamma \vdash M_1.M_2 : \mathsf{Cap}[\Sigma]}$$

$$\text{(OPEN)}\ \frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma]}{\Gamma \vdash \mathsf{open}\ a : \mathsf{Cap}[\Sigma]} \qquad \text{(INOUT)}\ \frac{\Gamma \vdash M : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \lessdot\!\# \, \mathsf{Amb}[\Sigma] \quad (M' \in \{\mathsf{in}\ M, \mathsf{out}\ M\})}{\Gamma \vdash M' : \mathsf{Cap}[\Sigma']}$$

$$\text{(MESSAGE)}\ \frac{\Gamma \vdash a : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \lessdot\!\# \, \mathsf{Amb}[\ \ell(\mathscr{V}')\ ] \quad \Gamma \vdash M' : \mathscr{V}'}{\Gamma \vdash a\ \mathsf{send}\ \ell\langle M' \rangle : \mathsf{Cap}[\Sigma]}$$

Processes

$$\text{(PREF)}\ \frac{\Gamma \vdash M : \mathsf{Cap}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash M.P : \mathsf{Proc}[\Sigma]} \qquad \text{(PAR)}\ \frac{\Gamma \vdash P : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash Q : \mathsf{Proc}[\Sigma]}{\Gamma \vdash P \mid Q : \mathsf{Proc}[\Sigma]}$$

$$\text{(RESTR)}\ \frac{\Gamma, x{:}\mathscr{A} \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash (\boldsymbol{\nu} x{:}\mathscr{A})P : \mathsf{Proc}[\Sigma]} \qquad \text{(DEAD)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \mathsf{Proc}[\Sigma]}$$

$$\text{(AMB)} \quad (\Sigma = (\ \ell_i(\mathscr{V}_i)\ )^{i \in I})$$
$$\frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma] \quad \Gamma, Z \lessdot\!\# \, \mathsf{Amb}[\Sigma], z{:}Z, x_i{:}\mathscr{V}_i \vdash P_i : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash a[(\ell_i(x_i) \triangleright \varsigma(z)P_i)\ ^{i \in I}\ ;\ P] : \mathsf{Proc}[\Sigma']}$$