Subtyping Mobile Classes and Mixins*

(extended abstract)

Lorenzo Bettini[1] Viviana Bono[2] Betti Venneri[1]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze, {bettini,venneri}@dsi.unifi.it
[2]Dipartimento di Informatica, Università di Torino, bono@di.unito.it

Abstract. MoMi (Mobile Mixins) is a coordination language for mobile processes that communicate and exchange object-oriented code in a distributed context. MoMi's key idea is structuring mobile object-oriented code by using mixin-based inheritance. Mobile code is compiled and typed locally, and can successfully interact with code present on foreign sites only if its type is subtyping-compliant with what is expected by the receiving site. In this paper, we study a subtyping relation for MoMi that includes both width subtyping and depth subtyping, in order to achieve a significantly more flexible, yet still simple, communication pattern. Technical problems arising from the depth subtyping are solved by defining a static annotation procedure.

1 Introduction

In [6], we introduced MoMi (Mobile Mixins), a coordination language for mobile processes that exchange object-oriented code. The underlying idea motivating MoMi is that standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not appear to scale well to a distributed context with mobility. MoMi's approach consists in structuring mobile object-oriented code by using mixin-based inheritance (a mixin is an incomplete class parameterized over a superclass, see [9, 2, 15]), and this is shown to fit into the dynamic and open nature of a mobile code scenario. For example, a downloaded mixin, describing a mobile agent that has to access some files, can be completed with a base class in order to provide access methods that are specific of the local file system. Conversely, critical operations of a mobile agent, enclosed in a downloaded class, can be redefined by applying a local mixin to it (e.g., in order to restrict the access to sensible resources, as in a *sand-box*). Therefore, MoMi is a clean combination of a core coordination calculus and an object-oriented mixin-based calculus.

The design of MoMi is illustrated and motivated with several examples in [6], and the calculus is completely formalized and its main properties are proved in [5]. The most important feature of MoMi's typing is the *subtyping* relation that guarantees safe, yet flexible, code communication. We assume that the code that is sent around has been successfully compiled, and it travels together with its static type. When the code is received on a site (whose code has been successfully compiled too) it is accepted only if its type is compliant with respect to the one expected, where compliance is based on subtyping. If the code is successfully accepted, it can interact with the local code in a safe way (i.e., no run-time errors) without requiring any further type checking of the whole code.

* This work has been partially supported by EU within the FET - Global Computing initiative, project AGILE IST-2001-32747 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

Thus, dynamic type checking is performed only at communication time. This is a crucial matter for mobility, since mobile code and in particular mobile agents are expected to be autonomous: once the communication successfully occurred, transmitted code behaves remotely in a (type) safe way (no failure communication to the sender will ever be required). This makes the code exchange an *atomic* action.

In our setting, the modalities of interaction with local code, are essentially object creations and mixin applications (in order to create hierarchies of classes). Thus, the subtyping relation involves not only object subtyping, but also a form of class subtyping and mixin subtyping: therefore, we provide subtyping hierarchies along the inheritance hierarchies. It is important to notice that we are not breaking brutally the design rule of keeping inheritance and subtyping separated, since only object subtyping is used when the code is executed locally. Instead, mixin and class subtyping plays a pivotal role during the communication, when classes and mixins become genuine run-time polymorphic values that can be transferred and computed by processes at a higher-order level.

The key novelty of the present work is the extension of the width subtyping for record types of [6, 5] with depth subtyping in order to achieve a more powerful mixin and class subtyping, and so increase communication flexibility in a substantial way. As a little informal example, think of someone who wants to download a videogame update. If the player is an expert, she surely knows where to find the right package with the right release, instead if the player is a beginner, she might find convenient to download a package: (*i*) containing more material than what she is looking for (width subtyping); (*ii*) including whatever release of the update is available (depth subtyping), and still she would be able to plug the update in her videogame.

There is general evidence that a flexible communication mechanism is of paramount importance in a distributed mobile code setting: on one hand, it is important to be "open" enough to accept utilities and services that are not exactly as we would have dreamed of (after all if we wanted something perfectly customized we would have written it ourselves, not searched for it on the net), on the other hand, it is also important to be "wise", that is, being able to single out possibly evil utilities or services from the ones that are harmless. We believe that our subtyping relation combining width subtyping and depth subtyping is a good step towards becoming "open" and "wise".

However, the proposed subtyping extension is far from being trivial technically. In fact, depth subtyping raises problems that mirror the very well known "depth subtyping versus override" problem in the object-based setting [1, 14, 7, 22]. Classes and mixins play in MoMi two orthogonal roles: they have their usual role as class hierarchies "builders" (so they rule the override mechanism), but at communication time they assume a polymorphic nature due to subtyping. This double nature brings some troubles that are specific of code mobility, where it is not predictable how a mobile class will be used when transmitted to different remote contexts.

The main contribution of this paper is an *annotation algorithm* that solves the "depth subtyping versus override" problem in a mobile mixin-based setting. Namely, the algorithm, while performing the local (static) type analysis of processes, derives suitable type constraints (lower bounds) for class types. This type information will be used dynamically at communication time, by the subtyping matching algorithm.

The paper outline is as follows. We present the syntax of MoMi in Section 2, and the type inference system in Section 3. Section 4 introduces the novel subtyping relation and Section 5 illustrates the "depth subtyping versus override" problem and the annotation algorithm. Section 6 concerns the operational semantics and Section 7 addresses some related works and future developments. Concerning the metatheory of the system here presented, we state only the main properties, omitting their proofs and intermediate results, for lack of space.

2 MoMi: Mobile Mixin Calculus

In this section we introduce the calculus MoMi. We first describe the syntax of the code exchanged among distributed mobile processes, which is object-oriented and structured upon mixin-based inheritance, as hinted in the introduction. We then present the coordination part, which includes representative features for distribution, communication and mobility of processes and code.

The object-oriented part of MoMi is defined as a standard class-based object-oriented language supporting mixin-based class hierarchies via *mixin definition* and *mixin application*. It was initially inspired by the core calculus of [8], and this is especially recognizable in MoMi's early version [6]. However, specific incarnations of most object-oriented notions (such as, e.g., functional or imperative nature of method bodies, object references, cloning, etc.) are irrelevant in this context, where the emphasis is on the structure of the object-oriented mobile code. Hence, we work here with a basic syntax forming the kernel calculus Sool (*Surface Object-Oriented Language*, shown in Table 1), including the essential features a language must support to be the MoMi's object-oriented component.

exp	$::=$	v	(value)
	$\mid$	$\mathsf{new}\ exp$	(object creation)
	$\mid$	$exp \Leftarrow m$	(method call)
	$\mid$	$v \diamond exp$	(mixin appl.)
v	$::=$	$\{m_i = f_i\ ^{i \in I}\}$	(record)
	$\mid$	x	(variable)
	$\mid$	$\mathsf{class}\ [m_i = f_i\ ^{i \in I}]\ \mathsf{end}$	(class def)
	$\mid$	mixin $\mathsf{expect}[m_i : \tau_{m_i}\ ^{i \in I}]$ $\mathsf{redef}[m_k : \tau_{m_k}\ \mathsf{as}\ \tau'_{m_k}\ \mathsf{with}\ f_k\ ^{k \in K}]$ $\mathsf{def}[m_j = f_j\ ^{j \in J}]$ end	(mixin def)

Table 1. Syntax of Sool.

Sool expressions offer object instantiation, method call and *mixin application* (to build class hierarchies); $\diamond$ denotes the mixin application operator and it always associates to the right. A Sool value, to which an expression reduces, is an object, which is essentially a recursive record $\{m_i = f_i\ ^{i \in I}\}$, a class definition, or a mixin definition, where $[m_i = f_i\ ^{i \in I}]$ denotes a sequence of method definitions, $[m_k : \tau_{m_k}\ \mathsf{as}\ \tau'_{m_k}\ \mathsf{with}\ f_k\ ^{k \in K}]$ denotes a sequence of method redefinitions, where τ_{m_k} is the type of the original method in the superclass and τ'_{m_k} is the new (smaller) type of the redefined method m_k. I, J and K are sets of indexes. Method bodies, denoted here with f (possibly with subscripts), are closed terms/programs and we abstract away from their actual form. In order to keep Sool simple, we decided to make expected methods' names and types and redefined methods' types explicit, although their types could be inferred (see, e.g., [8]). This choice does not harm the generality of MoMi. Another assumption we make is that methods do not accept/return classes and mixins as parameters/results (we will briefly hint why in the conclusions).

A mixin is essentially an abstract class that is parameterized over a (super)class. Let us describe informally the mixin use through a tutorial example:

$$\begin{array}{l} \mathrm{M} = \mathsf{mixin} \\ \quad \mathsf{expect}\ [n : \tau] \\ \quad \mathsf{redef}\ [m_2 : \tau_2\ \mathsf{as}\ \tau'_2\ \mathsf{with}\ \ldots\ next()\ \ldots] \\ \quad \mathsf{def}\ [m_1 = \ldots n() \ldots] \\ \mathsf{end} \end{array} \qquad \begin{array}{l} \mathrm{C} = \mathsf{class} \\ \quad [n = \ldots \\ \quad\ m_2 = \ldots] \\ \mathsf{end} \end{array} \qquad (\mathsf{new}\ (\mathrm{M} \diamond \mathrm{C})) \Leftarrow m_1()$$

Each mixin consists of three parts: (i) methods defined in the mixins, like m_1; (ii) *expected methods*, like n, that must be provided by the superclass; (iii) *redefined methods*, like m_2, where *next* can be used to access the (old) implementation of the method in the superclass. The application $M \diamond C$ constructs a class, which is a subclass of C.

$P ::=$	$\mathbf{nil}$	(null process)
$\mid$	$a.P$	(action prefixing)
$\mid$	$P_1 \mid P_2$	(parallel comp.)
$\mid$	X	(process variable)
$\mid$	$\mathsf{def}\ x = exp\ \mathsf{in}\ P$	(def)
$a ::=$	$\mathsf{send}(A, \ell)$	(send)
$\mid$	$\mathsf{receive}(id : \tau)$	(receive)
$A ::=$	$v \mid P$	(send's arg.)
$id ::=$	$x \mid X$	(receive's arg.)
$N ::=$	$\ell :: P$	(node)
$\mid$	$N_1 \parallel N_2$	(net composition)

Table 2. MoMi syntax.

MoMi's coordination component is similar to CCS [19] but also widely inspired by Klaim [12], since physical nodes are explicitly denoted as localities. MoMi is higher-order in that processes can be exchanged as first-entity data. A node is denoted by its locality, ℓ, and by the processes P running on it, i.e., $\ell :: P$. Informally, $\mathsf{send}(A, \ell)$ sends A, that can be either a process, P, or code represented as an object-oriented value, v, to locality ℓ, where there may be a process waiting for it by means of a $\mathsf{receive}$. The argument of $\mathsf{receive}$, id, ranges over x (a variable of Sool) and X (a process variable): for instance, a process can receive a class and integrate it into its local class hierarchy, or a process and spawn it for local execution. Our calculus is *synchronous*, but designing an asynchronous version would be straightforward.

We introduce the construct $\mathsf{def}\ x = exp\ \mathsf{in}\ P$ in order to pass to the sub-process P the result of the evaluation of an exp. In the processes $\mathsf{receive}(id : \tau).P$ and $\mathsf{def}\ x = exp\ \mathsf{in}\ P$, $\mathsf{receive}$ and def act as binders for, respectively, id and x in the process P. The $\mathsf{receive}$ action specifies, together with the formal parameter name, the type of the expected actual parameter. Notice that this is the only explicitly typed expression really needed in MoMi.

3 Typing

In this section we present the type system, starting from Sool, then we introduce the rules for typing processes.

$$
\begin{aligned}
\tau ::= &\ \Sigma \\
\mid &\ \mathsf{class}\langle \Sigma \rangle \\
\mid &\ \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle \\
\Sigma ::= &\ \{m_i : \tau_{m_i}\ ^{i \in I}\}
\end{aligned}
$$

Table 3. Syntax of types.

The set $\mathcal{T}$ of types is defined in Table 3. Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i}\ ^{i \in I}\}$. A record type can be viewed as a set of pairs *label*:*type* where labels are pairwise disjoint (Σ_1 and Σ_2 are considered *equals*, denoted by $\Sigma_1 = \Sigma_2$, if they differ only for the order of their elements). If $m_i : \tau_{m_i} \in \Sigma$ we say that the *subject* m_i *occurs* in Σ. $Subj(\Sigma)$ is the set of all subjects occurring in Σ. As we left method bodies unspecified (see Section 2), we must assume that there is a type system for the underlying part of Sool that types correctly method bodies, records, and some sort of fix-point. We will denote this type derivability with $\Vdash$, i.e., we will write $\Gamma \Vdash f : \tau$ and $\Gamma \Vdash \{m_i = f_i\ ^{i \in I}\} : \{m_i : \tau_{m_i}\ ^{i \in I}\}$. Rules for $\Vdash$ are obviously left implicit and $\Vdash$-statements are used as assumptions in other typing rules. Sool *typing environments* are sets of assumptions of the form $x : \tau$ and $m : \tau$, where x is a variable and m is a method name. As it is standard, $\Gamma, \bigcup_{i \in I} m_i : \tau_{m_i}$ (resp. $\Gamma, x : \tau$) will denote the environment obtained by adding to Γ all the assumptions $m_i : \tau_{m_i}$ (resp. the assumption $x : \tau$), provided that the resulting environment is still a typing environment (i.e., typing assumptions concern distinct variables and method names).

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (proj) \qquad \frac{}{\Gamma, m : \tau \vdash m : \tau}\ (proj) \qquad \frac{\Gamma \Vdash \{m_i = f_i \ ^{i \in I}\} : \{m_i : \tau_{m_i} \ ^{i \in I}\}}{\Gamma \vdash \{m_i = f_i \ ^{i \in I}\} : \{m_i : \tau_{m_i} \ ^{i \in I}\}}\ (rec)$$

$$\frac{\Gamma \vdash \{m_i = f_i \ ^{i \in I}\} : \{m_i : \tau_{m_i} \ ^{i \in I}\}}{\Gamma \vdash \mathsf{class}\ [m_i = f_i \ ^{i \in I}]\ \mathsf{end} : \mathsf{class}\langle\{m_i : \tau_{m_i} \ ^{i \in I}\}\rangle}\ (class)$$

$$\frac{\begin{array}{c} \Gamma, \bigcup_{i \in I} m_i : \tau_{m_i}, \bigcup_{k \in K} m_k : \tau'_{m_k} \vdash \{m_j = f_j \ ^{j \in J}\} : \{m_j : \tau_{m_j} \ ^{j \in J}\} \\ \Gamma, \bigcup_{i \in I} m_i : \tau_{m_i}, \bigcup_{k \in K} m_k : \tau'_{m_k}, \bigcup_{j \in J} m_j : \tau_{m_j}, next : \tau_{m_r} \Vdash f_r : \tau''_{m_r} \quad \tau''_{m_r} <: \tau'_{m_r} \quad \forall r \in K \\ Subj(\Sigma_{new}) \cap Subj(\Sigma_{exp}) = \emptyset \quad Subj(\Sigma_{new}) \cap Subj(\Sigma_{red}) = \emptyset \quad Subj(\Sigma_{red}) \cap Subj(\Sigma_{exp}) = \emptyset \\ \tau'_{m_k} <: \tau_{m_k} \quad \forall k \in K \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{mixin} \\ \ \mathsf{expect}[m_i : \tau_{m_i} \ ^{i \in I}] \\ \ \mathsf{redef}[m_k : \tau_{m_k}\ \mathsf{as}\ \tau'_{m_k}\ \mathsf{with}\ f_k \ ^{k \in K}] \\ \ \mathsf{def}[m_j = f_j \ ^{j \in J}] \\ \mathsf{end} \end{array} : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle}\ (mixin)$$

$$\text{where} \quad \begin{array}{l} \Sigma_{new} = \{m_j : \tau_{m_j} \ ^{j \in J}\}, \Sigma_{red} = \{m_k : \tau'_{m_k} \ ^{k \in K}\} \\ \Sigma_{exp} = \{m_i : \tau_{m_i} \ ^{i \in I}\}, \Sigma_{old} = \{m_k : \tau_{m_k} \ ^{k \in K}\} \end{array}$$

Table 4. Typing rules for SOOL values

The typing rules for SOOL values are in Table 4: class types $\mathsf{class}\langle\Sigma\rangle$ and mixin types $\mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle$ are formed over record types. A class type includes the type of its methods $\{m_i : \tau_{m_i} \ ^{i \in I}\}$. Mixin types encode the following information:

- Σ_{new}, Σ_{red} are the types of mixin methods (new and redefined, respectively).
- Σ_{exp}, Σ_{old} are the expected types of the methods that must be supported by any class to which the mixin is applied. Methods in Σ_{exp} are the methods that are not redefined by the mixin but still expected to be supported by the superclass since they may be called by other mixin methods, and methods in Σ_{old} are the types assumed for the superclass bodies of the methods redefined in the mixin. We refer to both sets of types as *expected* types since the actual superclass methods may have different types.
- Well typed mixins are well formed in the sense that name clashes among the different families of methods do not happen (the three clauses on subjects of the (*mixin*) rule) and that methods are redefined with subtypes of the old types (last clause).

The typing rules for SOOL expressions are in Table 5. Rule (*mixin app*) relies strongly on a subtyping relation $<:$ whose judgments are of the form $\tau_1 <: \tau_2$. This subtyping relation depends obviously on the nature of the SOOL calculus we choose (for instance, if SOOL is a functional calculus the subtyping relation may have the standard rule for the co/contra-variance behavior of the arrow type) but as an essential constraint it must contain the *width and depth subtyping* rule for record types. Our specimen record subtyping rule is an algorithmic subtyping rule as the one in [20]:

$$\frac{J \subseteq I \qquad \tau_{m_j} <: \tau'_{m_j} \ \forall j \in J}{\{m_i : \tau_{m_i} \ ^{i \in I}\} <: \{m_j : \tau'_{m_j} \ ^{j \in J}\}}\ (width\text{-}depth)$$

$$\frac{\Gamma \vdash exp : \{m_i : \tau_{m_i}\ ^{i \in I}\} \quad j \in I}{\Gamma \vdash exp \Leftarrow m_j : \tau_{m_j}}\ (lookup)$$	$$\frac{\Gamma \vdash exp : \mathsf{class}\langle\{m_i : \tau_{m_i}\ ^{i \in I}\}\rangle}{\Gamma \vdash \mathsf{new}\ exp : \{m_i : \tau_{m_i}\ ^{i \in I}\}}\ (new)$$

$$\frac{\begin{array}{l} \Gamma \vdash v : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle \\ \Gamma \vdash exp : \mathsf{class}\langle \Sigma_b \rangle \\ \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{old}) \\ \Sigma_{red} <: \Sigma_b / \Sigma_{red} \\ Subj(\Sigma_b) \cap Subj(\Sigma_{new}) = \emptyset \end{array}}{\Gamma \vdash v \diamond exp : \mathsf{class}\langle \Sigma_d \rangle}\ (mixin\ app)$$

$$\text{where} \quad \begin{array}{rl} \Sigma_d &= \Sigma_b/\Sigma_{exp} \cup \Sigma_{new} \cup \Sigma_{red} \cup \Sigma_{rest} \\ \Sigma_{rest} &= (\Sigma_b - (\Sigma_b/\Sigma_{exp} \cup \Sigma_b/\Sigma_{old})) \end{array}$$

Table 5. Typing rule for SOOL expressions.

Depth subtyping, as hinted in the introduction, is the main topic of the present paper; Section 5 will illustrate how to deal with the problems that depth subtyping introduces when it is used as the communication pattern.

In order to formalize the $(mixin\ app)$ rule, we introduce the following operation over record types ($m : \tau_1$ and $m : \tau_2$ are considered as distinct elements, and $\Sigma_1 \cup \Sigma_2$ and $\Sigma_1 - \Sigma_2$ are the standard set operations):

$$\Sigma_1/\Sigma_2 = \{m_i : \tau_{m_i} \mid m_i : \tau_{m_i} \in \Sigma_1 \ \wedge \ m_i \text{ occurs in } \Sigma_2\}$$

In the rule $(mixin\ app)$, Σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied. Then, Σ_b/Σ_{red} are the superclass methods redefined by the mixin, Σ_b/Σ_{exp} are the superclass methods called by the mixin methods but not redefined, and Σ_{rest} are the superclass methods not mentioned in the mixin definition at all. Notice that the superclass may have more methods than those required by the mixin constraints. The premises of the rule $(mixin\ app)$ are as follows:

i) $\Sigma_b <: (\Sigma_{exp} \cup \Sigma_{old})$ requires the actual types of the superclass methods be subtypes of those expected by the mixin.

ii) $\Sigma_{red} <: \Sigma_b/\Sigma_{red}$ requires that the actual implementation of methods in the superclass (which may belong to a subtype of the Σ_{old}) be supertypes of the ones redefined in the mixin. Thus, the types of the methods redefined by the mixin (Σ_{red}) will be subtypes of the superclass methods with the same name.

iii) $Subj(\Sigma_b) \cap Subj(\Sigma_{new}) = \emptyset$ guarantees that no name clash will take place during the mixin application.

Intuitively, the above constraints insure that all the actual method bodies of the newly created subclass are at least as "good" as the expected methods. The resulting class, of type $\mathsf{class}\langle\Sigma_d\rangle$, contains the signatures of all methods forming the new class created as a result of the mixin application. Σ_b/Σ_{exp} and Σ_{rest} are inherited directly from the superclass, Σ_{red} and Σ_{new} are defined by the mixin. Let us observe that, since for any well typed mixin $Subj(\Sigma_{red}) = Subj(\Sigma_{old})$, then, for any record type Σ, $\Sigma/\Sigma_{red} = \Sigma/\Sigma_{old}$.

Finally, we observe that rules defined in Tables 4 and 5 are syntax driven and do not use any explicit subsumption rule. Thus, assuming the basic type inference $\Vdash$ to be decidable, as well as $<:$, SOOL's typing turns out to be decidable too.

Typing rules for processes are defined in Table 6. At this stage we are not interested in typing processes in detail, so we will simply assign to a well-typed process the constant type proc. A process $\mathsf{receive}(X : \mathsf{proc}).X$ means that it is willing to receive any process and execute it. Observe that MoMi is a higher-order calculus where processes can exchange code that consists either of a process or of a SOOL value. Thus,

- the set $\mathcal{T}$ of types is extended to $\mathcal{T} \cup \{\mathsf{proc}\}$,
- type environments are extended with assertions $id : \tau$ where id ranges over x and X,
- type judgments are of the shape $\Gamma \vdash P : \mathsf{proc}$ (i.e., the process P is well typed from Γ),

$$\frac{}{\Gamma, X : \mathsf{proc} \vdash X : \mathsf{proc}}\ (proj) \qquad \frac{}{\Gamma \vdash \mathbf{nil} : \mathsf{proc}}\ (nil)$$

$$\frac{\Gamma \vdash A : \tau \qquad \Gamma \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{send}(A, \ell).P : \mathsf{proc}}\ (send) \qquad \frac{\Gamma, id : \tau \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{receive}(id : \tau).P : \mathsf{proc}}\ (receive)$$

$$\frac{\Gamma \vdash P_1 : \mathsf{proc} \qquad \Gamma \vdash P_2 : \mathsf{proc}}{\Gamma \vdash (P_1 \mid P_2) : \mathsf{proc}}\ (comp) \qquad \frac{\Gamma \vdash exp : \tau \qquad \Gamma, x : \tau \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{def}\ x = exp\ \mathsf{in}\ P : \mathsf{proc}}\ (def)$$

Table 6. Typing rules for processes

The rule (*send*) states that a process performing a send is well typed if both its argument and the continuation are well typed. For typing a process performing a $\mathsf{receive}$ we type the continuation with the information about the type of id (rule (*receive*)). Other rules are standard. Notice that if P has type proc then all object-oriented expressions occurring P are typed.

Finally, we will require that a process, in order to be executed on a site, must be closed (i.e., be without free variables), so it must be well-typed under $\Gamma = \emptyset$. It is easy to verify that if a process P is closed, then, for any $\mathsf{send}(A, \ell)$ occurring in P, the free variables of A are bound by an outer def or by an outer $\mathsf{receive}$. This implies that the exchanged code is closed when a send is executed. Decidability of typing defined in Table 6 trivially holds since applying rules bottom-up easily defines the algorithm (see Section 5).

4 Dynamic subtyping

The key idea of the present paper is the introduction of a novel subtyping relation, denoted by $\sqsubseteq$, defined on class and mixin types. This subtyping relation will be used to match dynamically the actual parameter's type against the formal parameter's type during communication. It is of paramount importance to notice that $\sqsubseteq$ is never used in the (local) static type inference. The operational semantics, where $\sqsubseteq$ is instead exploited, is presented in Section 6, but we anticipate here the introduction of $\sqsubseteq$ in order to motivate next section, where an annotation algorithm to be performed during the static analysis is presented. Such annotations, working together with $\sqsubseteq$, will ensure safe communications.

The subtyping relation $\sqsubseteq$ is defined in Table 7. The rule ($\sqsubseteq$ *class*) is naturally induced by the extended depth-and-width subtyping on record types. The rule ($\sqsubseteq$ *mixin*): (*i*) allows the subtype to define more new methods; (ii) prohibits to override more methods; (iii) allows a subtype to require fewer expected methods. Decidability of $\sqsubseteq$ follows trivially from decidability of $<:$.

$$\frac{\Sigma' <: \Sigma}{\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle}\ (\sqsubseteq\ class)$$

$$\frac{\Sigma'_{new} \supseteq \Sigma_{new} \qquad \Sigma_{exp} \supseteq \Sigma'_{exp}}{\mathsf{mixin}\langle\Sigma'_{new}, \Sigma_{red}, \Sigma'_{exp}, \Sigma_{old}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle}\ (\sqsubseteq\ mixin)$$

Table 7. Subtype on class and mixin types.

A common design principle for object-oriented programming languages is to keep the inheritance and the subtyping hierarchies completely separated so that subtyping only exists at the object level. Instead, in our mobile scenario, classes and mixins get a polymorphic and higher order nature during the mobile code exchange via $\sqsubseteq$. Thus, $\sqsubseteq$ can play the role of a rather flexible communication pattern: send and receive synchronize (see Table 11) if and only if the sent code is subtyping-compliant with the expected receive's parameter type. Informally speaking, it seems desirable to accept any class containing not only more resources but also more specialized resources than expected. Conversely, we would accept any mixin with fewer requests about methods expected from the superclass.

In order to achieve this communication mechanism, we need to annotate the send's argument with its type. However, this is not sufficient to guarantee a type-safe code exchange (i.e., that no "message-not-understood" error is produced after merging received code within the well-typed local code). We need to statically annotate programs with additional type information, as shown in the next section.

5 Annotating Processes

Let us consider a closed process P to be compiled. While reconstructing the derivation of $\emptyset \vdash P$:proc (obviously this derivation is unique, see the typing rules), it is easy to statically decorate any send argument occurring in P with its type. For instance, def $x = exp$ in send(x, ℓ) has type proc, and its compiled version is def $x = exp$ in send(x^{τ_1}, ℓ) if exp has type τ_1.

As briefly hinted in the introduction, the presence of depth subtyping conflicts with the overriding inheritance mechanism. First, we present an example (which is directly adapted from the one related to the object-based case of [1]). Let us consider the following expression:

$$\mathsf{receive}(x : \mathsf{class}\langle\{m : \mathsf{int}, n : \mathsf{int}\}\rangle).(\mathsf{new}\ M \diamond x) \Leftarrow m()$$

where M is a mixin redefining n with body -3. Now, receive could accept as an actual parameter a fully-fledged class $C : \mathsf{class}\langle\{m : \mathsf{int}, n : \mathsf{posint}\}\rangle$, where the actual body of m is $\log(self \Leftarrow n)$ (i.e., it invokes the sibling method n and applies the natural logarithm to the result of the invocation), since posint $<:$ int; however, the result of the execution of $(\mathsf{new}\ M \diamond C) \Leftarrow m()$ would raise a run-time error.

To abstract away from the details of the previous example, we consider the following situation: a variable $x : \mathsf{class}\langle\{m : \tau\}\rangle$ appearing in an expression of the form $M \diamond x$ and being the argument of a receive, with M a mixin that overrides $m : \tau_1$ with $\tau_1 <: \tau$. We might dynamically substitute to such x any received class $C : \mathsf{class}\langle\{m : \tau_2\}\rangle$, with $\mathsf{class}\langle\{m : \tau_2\}\rangle \sqsubseteq \mathsf{class}\langle\{m : \tau\}\rangle$, i.e., $\tau_2 <: \tau$. We can have three cases with respect to τ_1: (*i*) $\tau_1 <: \tau_2$; (*ii*) $\tau_2 <: \tau_1$; (*iii*) τ_1 and τ_2 are not comparable. The only case that does not create problems is case (*i*), the other ones are instead not type-safe. The conclusion is we have to make things in such a way that, when x is used in a mixin application as a class, it can be substituted only by actual classes with $\tau_1 <: \tau_2 <: \tau$.

As a consequence, the formal parameter of a receive, if it is of type "class", must be annotated not only with its explicit type (which acts as an upper bound for the type of the actual parameter), but also with some information about a "lower bound", such as the above $\tau_1 <: \tau_2$. This "lower bound", in general, cannot be simply another type because a "class" variable can appear inside a chain of mixin applications, and this may give rise to several constraints. Any receive argument of type "class" will be then annotated both with its type and with a type assertion $\mathfrak{A}$, which will contain no lower bound if the parameter does not participate in any mixin application.

Definition 1. *A* type assertion $\mathfrak{A}$ *is a property of the shape* $\mathfrak{A} = \inf(x : \mathsf{class}\langle\Sigma\rangle) : \Sigma'$, *where*

- Σ' *can be empty* ($\Sigma' = \emptyset$)
- $Subj(\Sigma') \subseteq Subj(\Sigma)$
- *if* $m : \tau' \in \Sigma'$ *then* $m : \tau \in \Sigma$ *with* $\tau' <: \tau$, *for some* τ.

Informally speaking, Σ' acts as an "inf" for Σ, since it contains lower bounds for some (possibly none) of the types associated to methods in Σ. We define $label(\mathfrak{A})$ as follows $label(\inf(x:\mathsf{class}\langle\Sigma\rangle) : \Sigma') = x$.

Definition 2. *Let* τ *be a class type,* $\tau = \mathsf{class}\langle\Sigma\rangle$, *and* $\mathfrak{A}$ *be a type assertion,* $\mathfrak{A} = \inf(x:\mathsf{class}\langle\Sigma_1\rangle) : \Sigma_2$. *We say that* τ satisfies $\mathfrak{A}$, *denoted by* $\tau \models \mathfrak{A}$, *if and only if*

- $\Sigma <: \Sigma_1$
- $\Sigma / \Sigma_2 :> \Sigma_2$.

In other words, $\mathsf{class}\langle\Sigma\rangle \models \inf(x : \mathsf{class}\langle\Sigma_1\rangle) : \Sigma_2$ means that Σ is a subtype of Σ_1, but for any method m such that $m : \tau \in \Sigma$ if $m : \tau_2 \in \Sigma_2$ then $\tau :> \tau_2$. Notice that if $\Sigma_2 = \emptyset$ the second condition trivially holds. For instance, the type $\tau = \mathsf{class}\langle\{m_1 : \tau_1, m_2 : \tau_2, m_3 : \tau_3\}\rangle$ satisfies the assertion $\mathfrak{A} = \inf(x : \mathsf{class}\langle\{m_1 : \tau_1^b, m_2 : \tau_2^b\}\rangle) : \{m_1 : \tau_1^{red}, m_2 : \tau_2^{red}\}$, provided that $\tau_1 <: \tau_1^b$ and $\tau_2 <: \tau_2^b$, and that $\tau_1^{red} <: \tau_1$ and $\tau_2^{red} <: \tau_2$.

We can collect assertions for several distinct variables, so obtaining a *type effect*.

Definition 3. *A* type effect $\mathcal{E}$ *is a set, possibly empty, of type assertions:* $\mathcal{E} = \{\mathfrak{A}_1, \ldots, \mathfrak{A}_n\}$, *where all* $\mathfrak{A}_i$ *are distinct, i.e.,* $label(\mathfrak{A}_i) \neq label(\mathfrak{A}_j)$, $1 \leq i, j \leq n$, *for* $i \neq j$.

Then we introduce the notion of *annotated processes*. An annotated process, denoted by $\overline{P}$, is a process decorated by adding: (*i*) types to the arguments of its sends; (*ii*) and types and type assertions to the arguments of its receives.

The procedure for annotating processes is described in two steps. Firstly, the algorithm Ann is defined on SOOL expressions: $Ann(\Gamma, exp)$ returns $\langle exp, \tau, \mathcal{E}\rangle$ where τ is the type of exp in Γ and $\mathcal{E}$ is a type effect. Then, we define $Ann(\Gamma, P)$ that returns $\langle\overline{P}, \mathsf{proc}, \mathcal{E}\rangle$; $\overline{P}$ is the annotated version of P, proc means that P is well typed in Γ and $\mathcal{E}$ is a type effect. Notice that, in both cases, the algorithm fails if the expression or the process are not typeable. However, in order to make the description of the algorithm simpler, we avoid handling failures explicitly.

$$
\begin{array}{l}
Ann(\Gamma, x) : \\
\quad \mathsf{let}\ \tau = \Gamma(x)\ \mathsf{in} \\
\quad \mathsf{if}\ \tau \equiv \mathsf{class}\langle \Sigma \rangle\ \mathsf{then} \\
\qquad \langle x, \tau, \{\inf(x : \mathsf{class}\langle \Sigma \rangle) : \emptyset\}\rangle \\
\quad \mathsf{else} \\
\qquad \langle x, \tau, \emptyset \rangle
\end{array}
\qquad
\begin{array}{l}
Ann(\Gamma, v) : \\
\quad \mathsf{let}\ \tau = \Gamma(v)\ \mathsf{in} \\
\qquad \langle v, \tau, \emptyset \rangle
\end{array}
\qquad
\begin{array}{l}
Ann(\Gamma, \mathsf{new}\ exp) : \\
\quad \mathsf{let}\ \langle exp, \mathsf{class}\langle \Sigma \rangle, \{\mathfrak{A}\}\rangle = Ann(\Gamma, exp)\ \mathsf{in} \\
\qquad \langle \mathsf{new}\ exp, \Sigma, \{\mathfrak{A}\}\rangle
\end{array}
$$

$$
\begin{array}{l}
Ann(\Gamma, exp \Leftarrow m) : \\
\quad \mathsf{let}\ \langle exp, \mathsf{class}\langle \{\ldots m : \tau \ldots\} \rangle, \{\mathfrak{A}\}\rangle = Ann(\Gamma, exp)\ \mathsf{in} \\
\qquad \langle exp \Leftarrow m, \tau, \{\mathfrak{A}\}\rangle
\end{array}
$$

$$
\begin{array}{l}
Ann(\Gamma, v \diamond exp) : \\
\quad \mathsf{let}\ \langle v, \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle, \emptyset \rangle = Ann(\Gamma, v)\ \mathsf{in} \\
\quad \mathsf{let}\ \langle exp, \mathsf{class}\langle \Sigma_b \rangle, \{\mathfrak{A}\}\rangle = Ann(\Gamma, exp)\ \mathsf{in} \\
\quad \mathsf{let}\ \Sigma_{rest} = (\Sigma_b - (\Sigma_b/\Sigma_{exp} \cup \Sigma_b/\Sigma_{old}))\ \mathsf{in} \\
\quad \mathsf{let}\ \Sigma_d = \Sigma_b/\Sigma_{exp} \cup \Sigma_{new} \cup \Sigma_{red} \cup \Sigma_{rest}\ \mathsf{in} \\
\quad \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{old}) \wedge \\
\quad \Sigma_{red} <: \Sigma_b/\Sigma_{red} \wedge \\
\quad Subj(\Sigma_{rest}) \cap Subj(\Sigma_{new}) = \emptyset \\
\qquad \langle v \diamond exp, \mathsf{class}\langle \Sigma_d \rangle, \{update(\mathfrak{A}, \Sigma_{red})\}\rangle
\end{array}
$$

Table 8. The annotation algorithm for expressions and values.

The algorithm Ann on expressions is in Table 8 and it is defined inductively on the structure of expressions. We observe that the type effect collected by the algorithm consists of one type assertion only, since a free variable of class type can occur at most once in a well-typed expression. When the algorithm is called on an identifier x of type $\mathsf{class}\langle \Sigma \rangle$, it creates a new assertion $\inf(x : \mathsf{class}\langle \Sigma \rangle) : \emptyset$ with no lower bound. When the algorithm is called on a mixin application $v \diamond exp$, no assertion can be generated by v; instead, Σ_{red} is used to generate a lower bound for updating the assertion generated by exp (the relation $\Sigma_{red} <: \Sigma_b/\Sigma_{red}$ is crucial for the type checking). Apart from class variables, the other values do not affect the resulting type assertion. The definition of the function *update* is as follows:

Definition 4. *Given an assertion $\mathfrak{A}$ and a record type Σ', $update(\mathfrak{A}, \Sigma')$ is the assertion $\mathfrak{A}'$ defined as follows:*

1. *if $\mathfrak{A} = \inf(x : \mathsf{class}\langle \Sigma \rangle) : \Sigma_1$ then $\mathfrak{A}' = \inf(x : \mathsf{class}\langle \Sigma \rangle) : \Sigma_1 \cup \Sigma_2$ where $\Sigma_2 = \{m_i : \tau_i \mid m_i : \tau_i \in \Sigma' \ \wedge \ m_i \notin Subj(\Sigma_1)\}$*
2. *otherwise, i.e. $\mathfrak{A} = \emptyset$, then $\mathfrak{A}' = \mathfrak{A}$*

Assertions are simply ignored if the rightmost expression in a mixin application is not an identifier, by definition of *update*. Moreover, *update* enriches the lower bound associated to a specific identifier only if a lower bound for a method had not already been defined; this guarantees that the greater lower bound for a redefined method is stored in the assertion.

The algorithm Ann for processes is in Table 9 and is defined inductively on the structure of processes or, equivalently, on typing rules for processes. The resulting $\mathcal{E}$ contains type assertions for identifiers bound by receives and occurring in the mixin application subterms of the process. When annotating a send, the argument A is annotated with its type, while the annotations generated for A are merged with the ones collected when annotating the continuation P. The identifier id of a receive is annotated with its type and with the assertion possibly generated for id during the annotation of the continuation ($\mathcal{E} \downarrow id$ is $\inf(id : \tau) : \Sigma$ if $\inf(id : \tau) : \Sigma \in \mathcal{E}$, and $\emptyset$ otherwise). Since $\mathsf{receive}$ is a binder for id it makes sense to discard the assertions for id from the type effect, after annotating the $\mathsf{receive}$ ($\mathcal{E} - (\mathcal{E} \downarrow id)$). The function *merge* takes two type effects and builds a new type effect.

Definition 5. *Given the types τ, τ_1 and τ_2 we define*

$$\tau_1 \sqcap_\tau \tau_2 = \begin{cases} \max(\tau_1, \tau_2) & \text{if } \tau_1 \text{ and } \tau_2 \text{ are comparable} \\ \tau & \text{otherwise} \end{cases}$$

Definition 6. *Given two type effects $\mathcal{E}_1$ and $\mathcal{E}_2$, $merge(\mathcal{E}_1, \mathcal{E}_2) = \mathcal{E}$, where $\mathcal{E}$ is defined as follows:*

- *for all $\mathfrak{A}_i \in \mathcal{E}_1$*
  - *if $label(\mathfrak{A}_i) \neq label(\mathfrak{A}_j)$ for all $\mathfrak{A}_j \in \mathcal{E}_2$ then $\mathfrak{A}_i \in \mathcal{E}$*
  - *else if $label(\mathfrak{A}_i) = label(\mathfrak{A}_j)$ for some $\mathfrak{A}_j \in \mathcal{E}_2$, let $\mathfrak{A}_i = \inf(x : \mathsf{class}\langle\Sigma\rangle) : \Sigma_i$ and $\mathfrak{A}_j = \inf(x : \mathsf{class}\langle\Sigma\rangle) : \Sigma_j$ then $\mathfrak{A}' \in \mathcal{E}$ where*
    $\mathfrak{A}' = \inf(x : \mathsf{class}\langle\Sigma\rangle) : \Sigma'$ *and*
    $\Sigma' = (\Sigma_i - \Sigma_j) \cup (\Sigma_j - \Sigma_i) \cup$
    $\{m : \tau' \mid m : \tau \in \Sigma, m : \tau_i \in \Sigma_i, m : \tau_j \in \Sigma_j, \tau' = \tau_i \sqcap_\tau \tau_j\}$
- *for all $\mathfrak{A}_k \in \mathcal{E}_2$ such that $label(\mathfrak{A}_k) \neq label(\mathfrak{A}_i)$ for all $\mathfrak{A}_i \in \mathcal{E}_1$, then $\mathfrak{A}_k \in \mathcal{E}$.*

$Ann(\Gamma, X)$:
  if $\mathsf{proc} = \Gamma(X)$ then
    $\langle X, \mathsf{proc}, \emptyset\rangle$

$Ann(\Gamma, \mathsf{send}(A, \ell).P)$:
  let $\langle \overline{A}, \tau, \mathcal{E}\rangle = Ann(\Gamma, A)$ in
  let $\langle \overline{P}, \mathsf{proc}, \mathcal{E}'\rangle = Ann(\Gamma, P)$ in
    $\langle \mathsf{send}(\overline{A}^\tau, \ell).\overline{P}, \mathsf{proc}, merge(\mathcal{E}, \mathcal{E}')\rangle$

$Ann(\Gamma, \mathsf{receive}(id : \tau).P)$:
  let $\langle \overline{P}, \mathsf{proc}, \mathcal{E}\rangle = Ann(\Gamma \cup \{id : \tau\}, P)$ in
    $\langle \mathsf{receive}(id^{\tau|(\mathcal{E}\downarrow id)}).\overline{P}, \mathsf{proc}, \mathcal{E} - (\mathcal{E} \downarrow id)\rangle$

$Ann(\Gamma, \mathsf{def}\ x = exp\ \mathsf{in}\ P)$:
  let $\langle exp, \tau, \{\mathfrak{A}\}\rangle = Ann(\Gamma, exp)$ in
  let $\langle \overline{P}, \mathsf{proc}, \mathcal{E}\rangle = Ann(\Gamma \cup \{x : \tau\}, P)$ in
    $\langle \mathsf{def}\ x = exp\ \mathsf{in}\ \overline{P}, \mathsf{proc}, merge(\{\mathfrak{A}\}, \mathcal{E})\rangle$

$Ann(\Gamma, P_1 \mid P_2)$:
  let $\langle \overline{P_1}, \mathsf{proc}, \mathcal{E}_1\rangle = Ann(\Gamma, P_1)$ in
  let $\langle \overline{P_2}, \mathsf{proc}, \mathcal{E}_2\rangle = Ann(\Gamma, P_2)$ in
    $\langle \overline{P_1} \mid \overline{P_2}, \mathsf{proc}, merge(\mathcal{E}_1, \mathcal{E}_2)\rangle$

Table 9. The annotation algorithm for processes.

Let us observe that a free variable can have many occurrences in a process P, in particular, if it has a class type then it can occur in many mixin application sub-expressions. This is the main difference between an expression and a process; for instance, in $P_1 \mid P_2$, a class variable can occur both in P_1 and P_2. Thus, during the annotation of a process, type assertions computed when annotating sub-processes are merged: if the sub-processes produce distinct type assertions, then both of them are collected; if they produce type assertions corresponding to the same variable, then the maximum lower bound for each method in the assertions is collected (Definition 5). A key property of *merge* is its *monotonicity*: merging two type effects never decreases the inf associated to variables' types.

Finally, the main property of *Ann* is its soundness w.r.t. the type inference, including a correctness and completeness property for type annotations. Informally, final annotations contain the greatest types among all types with which methods are redefined in mixin applications. This guarantees that substitution by narrowing class types is type-safe provided that those maximal lower bounds are respected. More formally:

Definition 7. *Given a mixin application expression* $M_n \diamond (\ldots \diamond (M_1 \diamond C) \ldots)$ *where* M_i *are mixin expressions having mixin types, respectively,* $\mathsf{mixin}\langle \Sigma^i_{new}, \Sigma^i_{red}, \Sigma^i_{exp}, \Sigma^i_{old} \rangle$*, and* C *is a class expression of type* $\mathsf{class}\langle \Sigma \rangle$*, we define:*

- *for any method* m *of* C *that is redefined by some* M_i *in such expression,* M_j *is said the* first redefinitor of m for C *iff* M_k *does not redefine* m*,* $\forall\ j-1 \leq k \leq 1$*;*
- $Top(M_n \diamond (\ldots \diamond (M_1 \diamond C) \ldots)) =$
  $\{m_i : \tau_i \mid m_i \in Subj(\Sigma), m_i : \tau_i \in \Sigma^j_{red}$ *and* M_j *is the first redefinitor of* m_i *for* $C\}$

Theorem 1 (Soundness). *If* $Ann(\Gamma, P) = \langle \overline{P}, \mathsf{proc}, \mathcal{E} \rangle$ *then*

i) $\Gamma \vdash P : \mathsf{proc}$
ii) for any free variable x *occurring in* P*, if* $x : \mathsf{class}\langle \Sigma \rangle$ *belongs to* Γ*, then there exists a type assertion* $\mathfrak{A} \in \mathcal{E}$ *such that* $\mathfrak{A} \equiv \inf(x : \mathsf{class}\langle \Sigma \rangle) : \Sigma'$ *and* $\Sigma' \supseteq Top(M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots))$ *for each* $M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots)$ *occurring in* P*.*

Theorem 2 (Type Safe Substitution). *Let* P *be a process such that* $\Gamma, x : \tau \vdash P : \mathsf{proc}$ *and* $Ann((\Gamma, x : \tau), P) = \langle \overline{P}, \mathsf{proc}, \mathcal{E} \rangle$*. Then, for any exp such that* $\Gamma \vdash exp : \tau_1$ *and* $\tau_1 <: \tau$ *(*$\tau_1 \sqsubseteq \tau$ *if they are class or mixin types), we have that* $\Gamma \vdash P[exp/x] : \mathsf{proc}$*, provided that the following condition is satisfied:*

(Cond)*: if* $\tau_1 \equiv \mathsf{class}\langle \Sigma \rangle$*, for some* Σ*, and* $label(\mathfrak{A}) = x$ *for some* $\mathfrak{A} \in \mathcal{E}$*, then* $\tau_1 \models \mathfrak{A}$*.*

Remark. One observation must be added to this discussion. Let us recall that the (*mixin app*) rule guarantees the absence of name clashes in any statically well typed expression of the shape $M \diamond C$. However, accidental overrides can occur when replacing at run-time M or C with M_1 and C_1 of smaller types, because of names of new methods possibly added by M_1 or C_1. This matter is related to the "width subtyping versus method addition" problem, that in our case boils down to a careful management of name clashes, a topic out of the scope of the present paper. Thus, we use [ ] for denoting a conventional capture-avoid-substitution, requiring possible renaming of methods with fresh names. From the point of view of the implementation (see the prototype implementation presented in [4]) this formal treatment of "global" fresh names can be solved with static binding for the mentioned methods (see [15] for an analogous treatment).

6 Operational Semantics

The operational semantics of MoMi involves two sets of rules. The first one describes how Sool object-oriented expressions reduce to values and is denoted by $\twoheadrightarrow$. We omit it here since it is quite standard. The second set of rules, presented in Table 11, describes the evolution of a MoMi net, showing how distributed processes communicate and exchange data and code by means of send and receive. It is based on a standard structural congruence $\equiv$ (defined as the least congruence relation closed under the rules in Table 10).

Notice that the semantics is defined on annotated (compiled) processes $\overline{P}$. The crucial point is the dynamic matching of types. send and receive synchronize only if the type of the delivered expression *matches* the one expected according to the following matching predicate:

$$N_1 \parallel N_2 = N_2 \parallel N_1$$
$$(N_1 \parallel N_2) \parallel N_3 = N_1 \parallel (N_2 \parallel N_3)$$
$$\ell :: \overline{P} = \ell :: \overline{P} \mid \mathbf{nil}$$
$$\ell :: (\overline{P_1} \mid \overline{P_2}) = \ell :: \overline{P_1} \parallel \ell :: \overline{P_2}$$

Table 10. Congruence laws

$$match_{\mathfrak{A}}(\tau_1, \tau_2) = \begin{cases} \tau_1 \models \mathfrak{A} & \text{if } \tau_1 \text{ and } \tau_2 \text{ are class types} \\ \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ and } \tau_2 \text{ are mixin types} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

The type τ_1 of the send's argument A is built statically by the annotation algorithm. The (*comm*) rule uses this type information, delivered together with the argument A, in order to dynamically check that the received item is correct w.r.t. the formal argument. In particular, if τ_1 is a class type, it is required to satisfy the type assertion $\mathfrak{A}$ that includes the constraint $\tau_1 \sqsubseteq \tau_2$ (Definition 2). The other rules are straightforward.

$$\frac{match_{\mathfrak{A}}(\tau_1, \tau_2)}{\ell_1 :: \mathsf{send}(\overline{A}^{\tau_1}, \ell_2).\overline{P'} \parallel \ell_2 :: \mathsf{receive}(id^{\tau_2 \mid \mathfrak{A}}).\overline{Q} \rightarrowtail \ell_1 :: \overline{P'} \parallel \ell_2 :: \overline{Q}[\overline{A}^{\tau_1}/id]} \ (comm)$$

$$\frac{exp \twoheadrightarrow v}{\ell :: \mathsf{def}\ x = exp\ \mathsf{in}\ \overline{P} \rightarrowtail \ell :: \overline{P}[v/x]} \ (def)$$

$$\frac{N_1 \rightarrowtail N_1'}{N_1 \parallel N \rightarrowtail N_1' \parallel N} \ (par) \qquad \frac{N \equiv N_1 \quad N_1 \rightarrowtail N_2 \quad N_2 \equiv N'}{N \rightarrowtail N'} \ (net)$$

Table 11. Net and process operational semantics

We assume that types are preserved under $\twoheadrightarrow$. Theorems 1 and 2 allow to prove that type proc is preserved under reduction by (*comm*) and then typing is preserved under $\rightarrowtail$. This easily extends to a global type safety for nets. A net N is *well typed* iff for any node $\ell :: \overline{P}$ in N, $\Gamma \vdash P : \mathsf{proc}$ for some Γ.

Theorem 3 (Subject Reduction). *If N is well typed and $N \rightarrowtail N'$ then N' is well typed.*

The key role of the above theorem consists in guaranteeing that merging code received from a remote site into local code does not harm local type safety. We observe that the dynamic checking during communication is the only dynamic use of types; it essentially consists in checking subtyping between record, class or mixin types, which is of linear complexity on the argument types. The type analysis of processes remains totally static and performed in each site independently.

Example Let us go back to the motivating example of the beginning of Section 5. The annotated versions of the processes there discussed are:

1. $\ell_1 :: \mathsf{send}(C^{\tau'}, \ell_2)$ where $\tau' = \mathsf{class}\langle\{m : \mathsf{int}, n : \mathsf{posint}\}\rangle$
2. $\ell_2 :: \mathsf{receive}(x^{\tau \mid \mathtt{inf}(x:\tau):\{n:\mathsf{int}\}}).(\mathsf{new}\ M \diamond x) \Leftarrow m()$ where $\tau = \mathsf{class}\langle\{m : \mathsf{int}, n : \mathsf{int}\}\rangle$

The communication between ℓ_1 and ℓ_2 cannot take place because $\tau' \not\models \mathrm{inf}(x : \tau) : \{n : \mathsf{int}\}$. In fact, $\tau' \sqsubseteq \tau$ but $\{n : \mathsf{int}\} \not<: \{m : \mathsf{int}, n : \mathsf{posint}\}/\{n : \mathsf{int}\}$, hence $match_{\mathtt{inf}(x:\tau):\{n:\mathsf{int}\}}(\tau', \tau)$ does not hold.

7 Conclusions

In the literature, there are several proposals of combining objects with processes and/or mobile agents. *Obliq* [11] is a lexically-scoped language providing distributed object-oriented computation. In [10], a general model for integrating object-oriented features in calculi of mobile agents is presented where agents are extended with constructs for remote method invocations. Other works, such as, e.g., [13, 21, 16] do not deal explicitly with mobile distributed code. Completely different, our approach addresses the question of a safe and scalable transmission of object-oriented code in a coordination calculus for mobile processes. The key role is played by the subtyping relation on classes and mixins, lifting type soundness of local code to a global type safety property, while ensuring a flexible communication.

MoMi is intended as a general framework that could be applied also to different mobile calculi; in particular, a prototype implementation of Klaim with MoMi's features is presented in [4]. Moreover, we are investigating how to extend MoMi and its type system with *incomplete objects* that should be dealt with likewise incomplete classes (mixins). Two other extensions may look interesting: (*i*) to introduce *higher-order mixins* as presented in [15], i.e., allowing also expressions of the shape $(M_1 \diamond M_2) \diamond C$; (*ii*) to allow methods to accept/return also classes and mixins. The first extension does not pose particular problems from the point of view of extending the annotation algorithm, the second one instead has to be carefully considered, in that not only variables of type "class" involved in a mixin application would need to be annotated, but also all variables of various nature standing for functions returning at some level something of type "class".

It is well-known that inheritance and synchronization constraints in a concurrent object-oriented setting often conflict: their simultaneous use tends to require many method redefinitions and to break encapsulation, so that typical practical advantages of object-oriented programming are lost. In [18] this phenomenon is studied and is given a name, *inheritance anomaly*. In MoMi, we do not provide explicit means for concurrency since we think that this is an orthogonal issue and should be dealt with by the underlying concrete language. However, if direct means for synchronization would be added to MoMi, either in the language underlying Sool or in the coordination language, we believe that some strategies could be adopted in order to avoid deadlocks and inheritance anomaly's drawbacks. We refer to [17, 3] for some techniques for avoiding these problems in concurrent object-oriented languages.

Acknowledgments. The authors wish to thank Mariangiola Dezani-Ciancaglini, Michele Boreale, and Paula Severi for comments on earlier drafts of this paper.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam - A Smooth Extension of Java with Mixins. In *ECOOP 2000*, number 1850 in LNCS, pages 145–178, 2000.
3. N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for $C^\sharp$. In B. Magnusson, editor, *Proc. of ECOOP02*, volume 2374 of *LNCS*, pages 415–440. Springer, 2002.
4. L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. Forthcoming.
5. L. Bettini, V. Bono, and B. Venneri. MoMi a calculus for mobile mixins. Internal Report, available at `http://music.dsi.unifi.it/papers.html`.

6. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In F. Arbarb and C. Talcott, editors, *Proc. of Coordination Models and Languages*, number 2315 in LNCS, pages 56–71. Springer, 2002.
7. V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL'94*, volume 933 of *LNCS*, pages 16–30. Springer, 1995.
8. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In R. Guerraoui, editor, *Proceedings ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer-Verlag, 1999.
9. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.
10. M. Bugliesi and G. Castagna. Mobile Objects. In *Proc. of FOOL*, 2000.
11. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
12. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
13. P. Di Blasio and K. Fisher. A Calculus for Concurrent Objects. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th Int. Conf.*, volume 1119 of *LNCS*, pages 655–670. Springer, 1996.
14. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *LNCS*, pages 42–61. Springer, 1995.
15. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
16. A. Gordon and P. Hankin. A Concurrent Object Calculus: Reduction and Typing. In U. Nestmann and B. C. Pierce, editors, *Proc. of HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*. Elsevier, 1998.
17. C. Laneve. Inheritance in Concurrent Objects. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing, An Object Oriented Approach*. Cambridge University Press, 2001.
18. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
19. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
20. B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.
21. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In T. Ito and A. Yonezawa, editors, *Proc. Theory and Practice of Parallel Programming (TPPP 94)*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.
22. J. Riecke and C. Stone. Privacy via Subsumption. *Information and Computation*, (172):2–28, 2002.