

# MOMI

## A Calculus for Mobile Mixins\*

Lorenzo Bettini<sup>1</sup>    Viviana Bono<sup>2</sup>    Betti Venneri<sup>1</sup>

<sup>1</sup>Dipartimento di Sistemi e Informatica, Università di Firenze, {bettini,venneri}@dsi.unifi.it

<sup>2</sup>Dipartimento di Informatica, Università di Torino, bono@di.unito.it

December 23, 2002

### 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 the type of what is expected by the receiving site. The key feature of the paper is the definition of this subtyping relation on classes and mixins that enables a significantly flexible, yet still simple, communication pattern. We show that communication by subtyping is type safe in that exchanged code is merged into local code without requiring further type analysis and recompilation.

## 1 Introduction

Internet provides technologies that allow the transmission of resources and services among computers distributed geographically in wide area networks. The growing use of a network as a primary environment for developing, distributing and running programs requires new supporting infrastructures. A possible answer to these requirements is the use of *mobile code* [35, 13] and in particular of *mobile agents* [27, 25, 38], which are software objects

---

\*This work has been partially supported by EU within the FET - Global Computing initiative, project MIKADO IST-2001-32222, DART project IST-2001-33477 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

consisting of data and code that can autonomously migrate to a remote computer and execute automatically on arrival.

Parallely, the object-oriented paradigm has become established as a well suited technology for designing and implementing large software systems. In particular it provides a high degree of modularity, then of flexibility and reusability, so that it is widely used also in distributed contexts (see, e.g., Java [3] and CORBA [31]).

The new scenario arising from mobility puts at test the *flexibility* of object-oriented code in a wider framework. Object-oriented components are often developed by different providers and may be downloaded on demand for being dynamically assembled with local applications. As a consequence, they must be strongly adaptive to any local execution environment, so that they can be dynamically reconfigured in several ways: downloaded code can be specialized by locally defined operations, and, conversely, locally developed software can be extended with new operations available from the downloaded code.

Hence a coordination language allowing to program object-oriented components as mobile processes should provide specific mechanisms for coordinating not only the transmission, but also the local dynamic reconfiguration of object-oriented code.

In this paper we address the above issue in the specific context of class-based languages, that “form the main stream of object-oriented programming” [1]. We propose to use a *mixin*-based approach for structuring mobile object-oriented code, as an alternative to standard inheritance mechanisms, as discussed in Section 2. A *mixin* (a class definition parameterized over the superclass) can be viewed as a function that takes a class as a parameter and derives a new subclass from it. The same mixin can be applied to many classes (the operation is known as *mixin application*), obtaining a family of subclasses with the same set of methods added and/or redefined. A subclass can be implemented before its superclass has been implemented; thus mixins remove most of the dependences of the subclass on the superclass, enabling dynamic development of class hierarchies. Mixins have become a focus of active research both in the software engineering [37, 34, 17] and programming language design [10, 36, 19] communities.

In our approach, we use mixins and mixin application as the mechanism for assembling mobile components in a flexible and safe way. To this aim, we propose a kernel language, MOMI (Mobile Mixins), integrating the mixin technology into a higher-order core calculus for mobile processes. Classes, mixins and objects, as well as processes, can be exchanged among different nodes, and classes and mixins can be integrated into different local class hierarchies. MOMI can be seen as a kernel language for programming and coordinating

network services that allow remote communication with transmission of object-oriented code and dynamic reconfiguration of class hierarchies. The calculus is equipped with a type system that guarantees static well typedness, since programs are compiled in each node independently. Then, code travels together with its static type and this type information can be dynamically used at communication time.

A novel subtyping relation on classes and mixins is introduced, in order to obtain a rather flexible communication pattern: two processes synchronize, by a send/receive mechanism, only if the sent code has a subtype of the one expected. This subtype is the key feature of MOMI. 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.

The subtyping here proposed guarantees both the above issues, flexibility and type safety: if the code is successfully accepted, after dynamic checking of subtyping, it can be merged into local code in a safe way without requiring any further recompilation and type analysis of the whole code. Thus, MOMI enjoys the benefits of static type checking and distributed processes use static types “dynamically” to coordinate themselves.

MOMI’s approach has been firstly presented in [6], where a preliminary version of the calculus is sketched. The present paper is an extended version of [6]: the syntax of the calculus is revised and, foremost, the type system is fully formalized and the main properties are proved.

The paper outline is as follows. In Section 2 we motivate our approach by investigating scenarios of object-oriented mobile code. We present the syntax of MOMI in Section 3, and the type system and the novel subtyping relation on classes and mixins in Section 4. In Section 5 we prove the main properties of typing and in particular the substitution property. Section 6 concerns the operational semantics and the subject reduction theorem on net evolution. In Section 7 we show an implementation of an example scenario in MOMI and Section 8 concludes the paper addressing some related works and future developments.

## 2 Mobility and Object-Oriented Code

In this section we discuss two different scenarios, where an object-oriented application is received from (sent to) a remote site. In this setting we can assume that the application consists of a piece of code  $A$  that moves to a remote site, where it will be composed with a local piece of code  $B$ . These scenarios may take place during the development of an object-oriented software system in a distributed context with mobility.

**Scenario 1** The local programmer may need to dynamically download classes in order to complete his own class hierarchy, without triggering off a chain reaction of changes over the whole system. For instance, he may want the downloaded class  $A$  to be a child class of a local class  $B$ . This generally happens in *frameworks* [22]: classes of the framework provide the general architecture of an application (playing the role of the local software), and classes that use the framework have to specialize them in order to provide specific implementations. The downloaded class may want to use operations that depend on the specific site (e.g., system calls); thus the local base class has to provide generic operations and the mobile code becomes a derived class containing methods that can exploit these generic operations.

**Scenario 2** The site that downloads the class  $A$  for local execution may want to redefine some, possibly critical, operations that remote code may execute. This way access to some sensitive local resources is not granted to untrusted code (for example, some destructive “read” operations should be redefined as non-destructive ones in order to avoid that non-trusted code erases information). Thus the downloaded class  $A$  is seen, in this scenario, as a base class, that is locally specialized in a derived class  $B$ .

Summarizing, in **1** the base class is the local code while in **2** the base class is the mobile code. These scenarios are typical object-oriented compositions seen in a distributed mobile context. A major requirement is that composing local code with remote code should not affect existing code in a massive way. Namely, both components and client classes should not be modified nor recompiled.

Standard mechanisms of class extension and code specialization would solve these design problems only in a static and local context, but they do not scale well to a distributed context with mobile code. The standard inheritance operation is essentially static in that it fixes the inheritance hierarchy, i.e., it binds derived classes to their parent classes, once for all at compile time. If such a hierarchy has to be changed, the program must be modified and then recompiled. This is quite unacceptable in a distributed mobile scenario,

since it would be against its underlying dynamic nature. Indeed, what we are looking for is a mechanism for providing a dynamic reconfiguration of the inheritance relation between classes, not only a dynamic implementation of some operations.

Let us go back and look in more details at the above scenarios. We could think of implementing some kind of “dynamic inheritance” for specifying at run-time the inheritance relation between classes without modifying their code. Such a technique could solve the difficulty raised by scenario **1**. However dynamic inheritance is not useful for solving scenario **2**, that would require a not so clear dynamic definition of the base class. Another solution would be releasing the requirement of not affecting the existing code, and allowing to modify the code of the local class (i.e., the local hierarchy). This could solve the second scenario, but not the first one that would require access to foreign source code. We are also convinced that the two scenarios should be dealt with by the same mechanism, allowing to dynamically use the same code in different environments, either as a base class for deriving new classes, or as derived class for being “adopted” by a parent class. We remark that a solution based on *delegation* could help solving these problems. However delegation would destroy at least the dynamic binding and the reusability of the whole system [8].

Summarizing, mobile object-oriented code needs to be much more flexible than locally developed applications. To this aim we propose a novel solution which is based on a mixin approach with subtyping and we show that it enables to achieve the sought dynamic flexibility. Indeed, mixin-based inheritance is more oriented to the concept of “completion” than to that of extendibility/specialization. Mixins are incomplete class specifications, parameterized over superclasses, thus the inheritance relation between a derived and a base class is not established through a declaration (e.g., like `extends` in Java), instead it can be coordinated by the operation of *mixin application*, that takes place during the execution of a program, and it is not in its declaration part.

### 3 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 above. We then present the coordination part, which includes representative features for distribution, communication and mobility of processes and code. MOMI is intended to be a small but general enough framework to experiment the integration of object-oriented features in several calculi for mobility and

distribution (such as, e.g., KLAIM, [14] and *DJoin*, [20]).

### 3.1 Surface Object-Oriented Language

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 [9], 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 a kernel calculus called here 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)
		$new\ exp$	(object instantiation)
		$exp \leftarrow m$	(method call)
		$v \diamond exp$	(mixin application)
$v$	$::=$	$\{m_i : \tau_{m_i} = f_i\}^{i \in I}$	(record)
		$x$	(variable)
		$class\ [m_i : \tau_{m_i} = f_i]^{i \in I}\ end$	(class def)
		$mixin$	(mixin def)
		$expect[m_i : \tau_{m_i}]^{i \in I}$	
		$redef[m_k : \tau_{m_k}\ with\ f_k]^{k \in K}$	
		$def[m_j : \tau_{m_j} = f_j]^{j \in J}$	
		$end$	

**Table 1:** Syntax of SOOL.

SOOL can be viewed as the contender for a minimal set of object-oriented features extended with mixin definitions and mixin applications. The design of SOOL favors compactness over completeness, so that it can be intended as the surface for a concrete underlying object-oriented language, similarly to the design of *Featherweight Java* [26].

SOOL expressions offer object instantiation, method call and *mixin application* (to built 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 record<sup>1</sup>  $\{m_i : \tau_{m_i} = f_i\}^{i \in I}$ , a class definition  $class\ [m_i : \tau_{m_i} = f_i]^{i \in I}\ end$ , or a mixin

<sup>1</sup>More precisely, we assume that an object is a recursive record where the self-references that are present in method bodies are solved by, for example, applying an appropriate fix-point operator, that binds *self*

definition `mixin expect` $[m_i : \tau_{m_i} \quad i \in I]$  `redef` $[m_k : \tau_{m_k} \text{ with } f_k \quad k \in K]$  `def` $[m_j : \tau_{m_j} = f_j \quad j \in J]$  `end`. In class and mixin definitions,  $[m_i : \tau_{m_i} = f_i \quad i \in I]$  is a sequence of method definitions (also abbreviated as  $\varrho$ ) and  $I$ ,  $J$  and  $K$  are sets of indexes. Method bodies, denoted here with  $f$  (possibly with subscripts), are closed terms/programs and they might have formal parameters (we abstract away from their actual form, relying on the form they may have in the concrete object-oriented language underlying SOOL).

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:

<pre>M = mixin   expect [n : τ]   redef [m<sub>2</sub> : τ<sub>2</sub> with ... next() ...]   def [m<sub>1</sub> : τ<sub>1</sub> = ... n() ...] end</pre>	<pre>C = class   [n : τ = ...    m<sub>2</sub> : τ<sub>2</sub> = ...] end</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------

$(\text{new } (M \diamond C)) \Leftarrow m_1()$

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 implementation of  $m_2$  in the superclass<sup>2</sup>. The application  $M \diamond C$  constructs a class, which is a subclass of  $C$ . Following the standard convention, class and mixin definitions are closed terms in that all free variables occurring in a method's body are bound to method's formal parameters. We do not consider explicit data fields, as they can be modeled in various ways.

We made the choice of explicitating: (i) names and types of the expected methods; and (ii) types of the defined and redefined methods. There are calculi/languages for which all this information is said to be inferred, for example in [9], so such names and types do not have to be explicitly given by the programmer. This would be acceptable from the point of view of MOMI, but we prefer to make for SOOL the simplest yet most general of the choices, since the focus is on using this type information for code mobility, rather than on deducing it.

### 3.2 Coordination language

MOMI's coordination component is similar to CCS [30] but also widely inspired by KLAIM [14], where physical nodes are explicitly denoted as localities. MOMI is higher-order in that processes can be exchanged as first-entity data.

---

to the host object, as it happens, for instance, in [9].

<sup>2</sup>We use the keyword “next” instead of the more common keyword “super” not to cause conflict with the term “super” referred to a class.

A node is denoted by its locality,  $\ell$ , and by the processes  $P$  running on it. Informally,  $\text{send}(A, \ell)$  sends  $A$ , that can be either a process,  $P$ , or code represented as an object-oriented value  $v$ , to locality  $\ell$ , where there may be a process waiting for it by means of a `receive`. The argument of `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.

In Table 2  $exp$  and  $v$  respectively denote an object-oriented expression and an object-oriented value, according to Table 1.

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

**Table 2:** MOMI syntax.

We introduce the construct `def  $x = exp$  in  $P$`  in order to pass to the sub-process  $P$  the result of the evaluation of an  $exp$ . In the processes `receive( $id : \tau$ ). $P$`  and `def  $x = exp$  in  $P$` , `receive` and `def` act as binders for, respectively,  $id$  and  $x$  in the process  $P$ . The `receive` action specifies, together with the formal parameter name, the type of the expected actual parameter. Notice that this is the only explicitly needed typed expression in the coordination language. In Section 4 a type system is introduced in order to assign a type to each well-behaved process.

### 3.3 Mixin Mobility in Action

We present in the following two simple examples showing mobility of mixins in action. They represent a *remote evaluation* and a *code-on-demand* [13] situation, respectively. Let us observe that both situations can be seen as examples of mobile agents as well. A more complex example is presented in Section 7.

**Example 1.** Let `agent` represent the type of a mixin defining a mobile agent that must print some data by using the local printer on any remote site where it is shipped for execution. Obviously, since the `print` operation highly depends on the execution site (even only because of the printer drivers), it is sensible to leave such method not implemented. The mixin then can be applied, on the remote site, to a local class `printer` which will provide the specific implementation of the `print` method in the following way:

```

ℓ1 :: ... | send(my_agent, ℓ2) ||
ℓ2 :: ... | receive(mob_agent : agent).
def PrinterAgent = mob_agent ◊ printer in
  (new PrinterAgent) ⇐ start()

```

**Example 2.** Let `agent` be a class defining a mobile agent that may access the file system of a remote site. If the remote site wants to execute this agent while restricting the access to its own file system, it can locally define a mixin `restricted`, redefining the methods accessing the file system according to specific restrictions. Then the arriving agent can be composed with the local mixin in the following way.

```

ℓ1 :: ... | send(my_agent, ℓ2) ||
ℓ2 :: ... | receive(mob_agent : agent).
def RestrictedAgent = restricted ◊ mob_agent in
  (new RestrictedAgent) ⇐ start()

```

This example can also be seen as an implementation of a “sandbox”.

The above examples highlight how an object-oriented expression (`mob_agent`) can be used by the receiver site both as a mixin (Example 1) and as a base class (Example 2). Indeed, without any change to the code of the examples, one could also dynamically construct a class such as `restricted ◊ mob_agent ◊ printer`. It is important to remark that in these examples we assume that the sent code (argument of `send`) and the expected code (argument of `receive`) are “compatible”. This will be guaranteed by the dynamic matching between the actual parameter type and the formal parameter type in the communication rule (see Table 10).

## 4 Typing

We present the type system for MOMI starting from SOOL, then we introduce the rules for typing processes. We conclude this section by introducing a novel subtyping relation,

$\sqsubseteq$ , over class and mixin types, which relies on the standard width subtyping over record types (as found in other languages such as Java and C++).

#### 4.1 Typing SOOL expressions

The set  $\mathcal{T}$  of types is defined in Table 3.  $\Sigma$  (possibly with a subscript) denotes a record type of the form  $\{m_i : \tau_{m_i} \mid i \in I\}$ . A record type can be viewed as a set of pairs *label:type*, with the additional requirement that labels are pairwise disjoint. Thus notations and operations on sets are easily extended to record types as in the following definitions:

- if  $m_i : \tau_{m_i} \in \Sigma$  we say that the *subject*  $m_i$  *occurs* in  $\Sigma$  (with type  $\tau_{m_i}$ ). Then,  $Subj(\Sigma)$  denotes the set of all subjects occurring in  $\Sigma$ ;
- $\Sigma_1 \subseteq \Sigma_2$  is the standard set inclusion, i.e.,  $m : \tau_m \in \Sigma_1 \Rightarrow m : \tau_m \in \Sigma_2$ ;
- $\Sigma_1$  and  $\Sigma_2$  are considered *equals*, denoted by  $\Sigma_1 = \Sigma_2$ , if  $\Sigma_1 \subseteq \Sigma_2$  and  $\Sigma_2 \subseteq \Sigma_1$  (i.e., they differ only for the order of their elements);
- $\Sigma_1 \cup \Sigma_2$  is the standard set union (used only on  $\Sigma_1$  and  $\Sigma_2$  such that  $Subj(\Sigma_1) \cap Subj(\Sigma_2) = \emptyset$  in order to guarantee  $\Sigma_1 \cup \Sigma_2$  to be a record type);
- $\Sigma_1 - \Sigma_2$  is the standard set difference;
- $\Sigma_1 / \Sigma_2 = \{m_i : \tau_{m_i} \mid m_i : \tau_{m_i} \in \Sigma_1 \wedge m_i \text{ occurs in } \Sigma_2\}$ .

$\begin{aligned} \tau & ::= \Sigma \\ & \quad   \text{class}(\Sigma) \\ & \quad   \text{mixin}(\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}) \\ \Sigma & ::= \{m_i : \tau_{m_i} \mid i \in I\} \end{aligned}$
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Table 3:** Syntax of types.

As we left method bodies unspecified (see Section 3), we must assume that there is a type system for the underlying part of SOOL, i.e., method bodies, records and objects. We will denote this typing with  $\Vdash$ , i.e., we will write  $\Gamma \Vdash f : \tau$  and  $\Gamma \Vdash \{m_i : \tau_{m_i} = f_i \mid i \in I\} : \{m_i : \tau_{m_i} \mid i \in I\}$ . Rules for  $\Vdash$  are obviously left implicit and  $\Vdash$ -statements are used as assumptions in other typing rules. SOOL *typing environments* are sets of typing 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  $\Gamma$  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).

Typing rules will strongly rely 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 a rule for the co/contra-variance behavior of the arrow type) but as an essential constraint it must contain the standard *width subtyping* rule on record types, that is:

$$\frac{\Sigma_2 \subseteq \Sigma_1}{\Sigma_1 <: \Sigma_2} \text{ (width)}$$

Informally speaking, (*width*) rule captures the intuition that the subtype is the one with more methods. Such rule for record types is the only one we need to make explicit here.

Class types  $\text{class}\langle \Sigma \rangle$  and mixin types  $\text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$  are formed over record types. A class type includes the type of its methods  $\{m_i : \tau_{m_i} \mid i \in I\}$ . Mixin types encode the following information:

- $\Sigma_{new}, \Sigma_{red}$  are the exact types of mixin methods (newly introduced, called from now on *new*, and redefined, respectively).
- $\Sigma_{exp}$  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.
- Well formed mixin types assume that name clashes among the different families of methods do not happen (this is checked by the typing rules).

The typing rules for SOOL values are in Table 4. The most interesting one is the (*mixin*) rule: The clause ( $c_1$ ) concerns the types of the new methods defined in the mixin and uses types assumptions about expected and redefined methods; the clause ( $c_2$ ) checks, for each redefined method, that its body is well typed using also the types of new methods, checked by ( $c_1$ ). Notice that the body of a redefined method is well typed when its type is a subtype of the type of the original method of the superclass, which is explicitly given in the mixin definition and thus assumed for *next*. Let us observe that, since depth subtyping is not defined in this calculus, a redefined method keeps the type of its old version (the one of the superclass) in the mixin type, even though it is sound to assume that the method body can typecheck “internally” with a smaller type (clause  $\tau'_{m_r} <: \tau_{m_r}$ ). Finally, ( $c_3$ ) checks the well-formedness of the mixin, i.e., that there is no name clashes among method names in record types  $\Sigma_{new}, \Sigma_{red}$  and  $\Sigma_{exp}$ .

$\frac{}{\Gamma, x : \tau \vdash x : \tau} \text{ (proj)}$	$\frac{}{\Gamma, m : \tau \vdash m : \tau} \text{ (proj)}$	$\frac{\Gamma \quad \{m_i : \tau_{m_i} = f_i^{i \in I}\} : \{m_i : \tau_{m_i}^{i \in I}\}}{\Gamma \vdash \{m_i : \tau_{m_i} = f_i^{i \in I}\} : \{m_i : \tau_{m_i}^{i \in I}\}} \text{ (rec)}$
$\frac{\Gamma \vdash \{m_i : \tau_{m_i} = f_i^{i \in I}\} : \{m_i : \tau_{m_i}^{i \in I}\}}{\Gamma \vdash \text{class}[m_i : \tau_{m_i} = f_i^{i \in I}] \text{ end} : \text{class}\langle\{m_i : \tau_{m_i}^{i \in I}\}\rangle} \text{ (class)}$		
<p>(c<sub>1</sub>) <math>\Gamma, \bigcup_{i \in I} m_i : \tau_{m_i}, \bigcup_{k \in K} m_k : \tau_{m_k} \vdash \{m_j : \tau_{m_j} = f_j^{j \in J}\} : \{m_j : \tau_{m_j}^{j \in J}\}</math></p> <p>(c<sub>2</sub>) <math>\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}, \text{next} : \tau_{m_r} \quad f_r : \tau'_{m_r} \quad \tau'_{m_r} &lt;: \tau_{m_r} \quad \forall r \in K</math></p> <p>(c<sub>3</sub>) <math>\text{Subj}(\Sigma_{new}) \cap \text{Subj}(\Sigma_{exp}) = \emptyset \quad \text{Subj}(\Sigma_{new}) \cap \text{Subj}(\Sigma_{red}) = \emptyset \quad \text{Subj}(\Sigma_{red}) \cap \text{Subj}(\Sigma_{exp}) = \emptyset</math></p>		
$\frac{}{\Gamma \vdash \text{mixin} \quad \text{expect}[m_i : \tau_{m_i}^{i \in I}] \quad \text{redef}[m_k : \tau_{m_k} \text{ with } f_k^{k \in K}] : \text{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \quad \text{def}[m_j : \tau_{m_j} = f_j^{j \in J}] \quad \text{end}} \text{ (mixin)}$		
<p>where <math>\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}\}</math></p>		

Table 4: Typing rules for SOOL values

$\frac{\Gamma \vdash \text{exp} : \{m_i : \tau_{m_i}^{i \in I}\} \quad j \in I}{\Gamma \vdash \text{exp} \leftarrow m_j : \tau_{m_j}} \text{ (lookup)}$	$\frac{\Gamma \vdash \text{exp} : \text{class}\langle\{m_i : \tau_{m_i}^{i \in I}\}\rangle}{\Gamma \vdash \text{new exp} : \{m_i : \tau_{m_i}^{i \in I}\}} \text{ (new)}$
$\Gamma \vdash v : \text{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle$	
$\Gamma \vdash \text{exp} : \text{class}\langle\Sigma_b\rangle$	
$\Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red})$	
$\text{Subj}(\Sigma_b) \cap \text{Subj}(\Sigma_{new}) = \emptyset$	
$\frac{}{\Gamma \vdash v \diamond \text{exp} : \text{class}\langle\Sigma_b \cup \Sigma_{new}\rangle} \text{ (mixin app)}$	

Table 5: Typing rules for SOOL expressions.

The typing rules for SOOL expressions are in Table 5. The premises of the rule (*mixin app*) (a syntactic variation of the corresponding rule of [9]) are as follows:

- i)  $\Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red})$  requires that the superclass provides all the methods that the mixin expects and redefines.
- ii)  $\text{Subj}(\Sigma_b) \cap \text{Subj}(\Sigma_{new}) = \emptyset$  guarantees that no name clash will take place during the mixin application.

Notice that the superclass may have more methods than those required by the mixin constraints. Thus, the type of the mixin application expression is a class type containing both the signatures of all the methods supplied by the superclass ( $\Sigma_b$ ) and those of the new methods defined by the mixin ( $\Sigma_{new}$ ).

## 4.2 Typing processes

In this section we present the type system for deciding whether a process is well typed. 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 `receive( $X : \text{proc}$ ). $X$`  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 (shown in Table 2). Thus,

- the set  $\mathcal{T}$  of types is extended to  $\mathcal{T} \cup \{\text{proc}\}$ ,
- we assume `proc`  $<$ : `proc`,
- type environments are extended with assertions  $id : \tau$  where  $id$  ranges over  $x$  and  $X$ ,
- type judgments are of the shape  $\Gamma \vdash P : \text{proc}$  saying that the process  $P$  is well typed from  $\Gamma$ .

Typing rules are defined as in Table 6.

$\frac{}{\Gamma, X : \text{proc} \vdash X : \text{proc}} \text{ (proj)}$	$\frac{}{\Gamma \vdash \mathbf{nil} : \text{proc}} \text{ (nil)}$
$\frac{\Gamma \vdash A : \tau \quad \Gamma \vdash P : \text{proc}}{\Gamma \vdash \text{send}(A, \ell).P : \text{proc}} \text{ (send)}$	$\frac{\Gamma, id : \tau \vdash P : \text{proc}}{\Gamma \vdash \text{receive}(id : \tau).P : \text{proc}} \text{ (receive)}$
$\frac{\Gamma \vdash P_1 : \text{proc} \quad \Gamma \vdash P_2 : \text{proc}}{\Gamma \vdash (P_1 \mid P_2) : \text{proc}} \text{ (comp)}$	$\frac{\Gamma \vdash \text{exp} : \tau \quad \Gamma, x : \tau \vdash P : \text{proc}}{\Gamma \vdash \text{def } x = \text{exp} \text{ in } P : \text{proc}} \text{ (def)}$

**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 `receive` we type the continuation with the information about the type of  $id$  (rule (*receive*)). Rule (*def*) is standard. For parallel composition (rule (*comp*)) we require that both processes have the same type `proc`. Notice that if  $P$  has type `proc` then all object-oriented sub-expressions of  $P$  are typed.

Let us observe that we do not use an explicit *subsumption* rule for  $<$ : in the type system, but rather the algorithmic discipline, where the use of  $<$ : is performed when needed. This has the important advantage that every typeable expression has a unique type. 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  $\text{send}(A, \ell)$  occurring in  $P$ , the free variables of  $A$  are bound by an outer  $\text{def}$  or by an outer  $\text{receive}$ . This implies that the exchanged code is closed when a  $\text{send}$  is executed.

### 4.3 Subtyping on class and mixin types

A 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 ( $\text{send}$ 's argument) against the formal parameter's type ( $\text{receive}$ 's argument) 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 state the substitution property, and to enucleate the crucial steps for proving the subject reduction theorem.

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

$\frac{\Sigma' <: \Sigma}{\text{class}\langle \Sigma' \rangle \sqsubseteq \text{class}\langle \Sigma \rangle} (\sqsubseteq \text{ class})$
$\frac{\Sigma'_{\text{new}} <: \Sigma_{\text{new}} \quad \Sigma_{\text{exp}} <: \Sigma'_{\text{exp}} \quad \Sigma'_{\text{red}} = \Sigma_{\text{red}}}{\text{mixin}\langle \Sigma'_{\text{new}}, \Sigma'_{\text{red}}, \Sigma'_{\text{exp}} \rangle \sqsubseteq \text{mixin}\langle \Sigma_{\text{new}}, \Sigma_{\text{red}}, \Sigma_{\text{exp}} \rangle} (\sqsubseteq \text{ mixin})$

**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 nature during the mobile code exchange via  $\sqsubseteq$ . From this perspective, MOMI can be viewed as a sort of tradeoff between the class-based approach and the object-based one: while object-oriented code is structured in a class-based way, mobile object-oriented code is exchanged among processes by a mechanism very similar to an object-based setting.

The relation  $\sqsubseteq$  plays the role of a communication pattern:  $\text{send}$  and  $\text{receive}$  synchronize (see Table 10) if and only if the sent code is subtyping-compliant with the expected  $\text{receive}$ 's parameter type. Informally speaking, one can receive any class containing more resources

than he expected. Conversely, we would accept any mixin with weaker requests about methods expected from the superclass.

The type system guarantees that if some code was statically type-checked on a site, it can be sent to a different site by means of the `send/receive` mechanism, and, if its type is subtyping-compliant with the expected parameter type, it will not produce any “message-not-understood” run-time error when merged within the local (well-typed) code, and executed. This crucial point is guaranteed by the subject reduction theorem in Section 6. In order to achieve this communication mechanism, we need to decorate the `send`’s argument with its type. This can be easily done during type checking in the static analysis, in order to produce *compiled processes* to be evaluated (see Definition 6.2).

## 5 Properties of typing

The most important property of MOMI’s type system is the *type safe substitution*, on which preservation of types during evaluation will rely (see Section 6). Informally speaking, a process of the shape `receive(id:τ).P` will accept any argument of type  $\tau_1$  if  $\tau_1$  is a subtype of  $\tau$ , where subtyping coincides with  $\sqsubseteq$  in case of mixin and class definitions. To this aim, we need to prove that  $P$  remains well typed after replacing free occurrences of `id` of type  $\tau$  with any received code having a subtype of  $\tau$ .

This substitution property will be formally given at the end of this section (Theorem 5.1), after proving few basic lemmas for dealing with crucial cases when `id` has a mixin or class type and it occurs in a mixin application expression. A second question concerns accidental name clashes that can occur during the above substitution based on subtyping. For a careful management of the name-avoid-capture substitution, we refer to Section 5.1.

It is easy to verify that a well-typed application of a mixin to a class generates a subtype of the class argument.

**Property 1** *If  $\Gamma \vdash v : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$ ,  $\Gamma \vdash exp : \text{class}\langle \Sigma_b \rangle$  and  $\Gamma \vdash v \diamond exp : \text{class}\langle \Sigma_d \rangle$ , then  $\text{class}\langle \Sigma_d \rangle \sqsubseteq \text{class}\langle \Sigma_b \rangle$ .*

**Proof:** By the definition of (*mixin app*) rule, (Table 5),  $(\Sigma_b \cup \Sigma_{new}) <: \Sigma_b$ . □

Decidability of MOMI’s type system, namely uniqueness of types, easily follows from the typing relation that defines a typing rule for each construct.

**Lemma 5.1 (Decidability)** *In a given context  $\Gamma$ , for any process  $P$  (with free variables all in the domain of  $\Gamma$ ) it is decidable whether  $\Gamma \vdash P : \text{proc}$ .*

**Proof:** Rules (*new*), (*lookup*) and (*mixin app*) are syntax driven and the subsumption rule is avoided, thus if an expression is typeable, then it is assigned a unique type. Concerning type environments, if  $exp$  has type  $\tau$  then there is a minimal environment  $\Gamma$  (assigning types to free variables of  $exp$ ) such that  $\Gamma \vdash exp : \tau$ . For any other environment  $\Gamma'$  such that  $\Gamma' \vdash exp : \tau$ , we have that  $\Gamma$  is included in  $\Gamma'$ . Then, assuming that  $<:$  is decidable, typeability of processes follows from the shape of rules in Table 6, which are syntax driven.  $\square$

For simplicity, in the following we will use  $M$  and  $C$  (possibly with subscripts) for denoting mixin definitions and class definitions, having mixin and class types respectively.

### 5.1 Substitution and name clashes

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 clashes can occur when replacing at run-time  $M$  or  $C$  with  $M_1$  or  $C_1$  having subtypes, 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 (well known in the object-based setting, see for instance [18]), that in our case boils down to a careful management of these *dynamic name clashes*. Thus, we have to define a suitable capture-avoid-substitution,  $[ ]$ , requiring possible renaming of methods with fresh names.

The basic idea is that, if  $x$  is of type  $\text{class}\langle\Sigma\rangle$ ,  $C$  is of type  $\text{class}\langle\Sigma'\rangle$  and  $\text{class}\langle\Sigma'\rangle \sqsubseteq \text{class}\langle\Sigma\rangle$ , method names are assumed fresh for all subjects belonging to  $\Sigma' - \Sigma$  in  $[C/x]$ . The case of mixin substitution is dealt with analogously, acting on  $\Sigma_{new}$ . Moreover, we have to be sure that a different refreshed version of a class or a mixin value is substituted for each occurrence of the replaced variable. In order to do that formally, we define a *refresh* function that allows to provide the actual substitution algorithm.

**Definition 5.1** *Let  $v$  be a class or a mixin definition and  $\mathcal{M}$  a set of method names such that methods in  $\mathcal{M}$  are methods defined in  $v$ ; we define  $\text{refresh}_{\mathcal{M}}(v)$  as the definition  $v'$  obtained from  $v$  by lexically renaming all the labels belonging to  $\mathcal{M}$  with fresh names.*

Then we define a substitution parameterized over a record type, which relies on the function *refresh*.

**Definition 5.2 (Parametric Substitution)** *Let  $v$  denote a class or mixin definition. Let  $P$  be a well typed process and  $x$  be a free variable of type class or mixin, respectively. For any set of method names  $\mathcal{M}$ ,  $P[v/x]_{\mathcal{M}}$  is defined by structural induction on  $P$  in the following way:*

- $\mathbf{nil}[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{nil}$
- $(P_1|P_2)[v/x]_{\mathcal{M}} \stackrel{def}{=} P_1[v/x]_{\mathcal{M}}|P_2[v/x]_{\mathcal{M}}$
- $X[v/x]_{\mathcal{M}} \stackrel{def}{=} X$
- $(\mathbf{def} \ y = \mathit{exp} \ \mathbf{in} \ P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{def} \ y = (\mathit{exp}[v/x]_{\mathcal{M}}) \ \mathbf{in} \ P[v/x]_{\mathcal{M}}$
- $(\mathbf{def} \ x = \mathit{exp} \ \mathbf{in} \ P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{def} \ x = (\mathit{exp}[v/x]_{\mathcal{M}}) \ \mathbf{in} \ P$
- $(\mathbf{receive}(y : \tau).P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{receive}(y : \tau).(P[v/x]_{\mathcal{M}})$
- $(\mathbf{receive}(x : \tau).P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{receive}(x : \tau).P$
- $(\mathbf{send}(A, \ell).P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{send}(A[v/x]_{\mathcal{M}}, \ell).(P[v/x]_{\mathcal{M}})$

where

- $y[v/x]_{\mathcal{M}} \stackrel{def}{=} y$
- $x[v/x]_{\mathcal{M}} \stackrel{def}{=} [\mathit{refresh}_{\mathcal{M}}(v)/x]$
- $(\mathbf{new} \ \mathit{exp})[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{new} \ (\mathit{exp}[v/x]_{\mathcal{M}})$
- $(\mathit{exp} \leftarrow m)[v/x]_{\mathcal{M}} \stackrel{def}{=} (\mathit{exp}[v/x]_{\mathcal{M}}) \leftarrow m$
- $(v_1 \diamond \mathit{exp})[v/x]_{\mathcal{M}} \stackrel{def}{=} (v_1[v/x]_{\mathcal{M}}) \diamond (\mathit{exp}[v/x]_{\mathcal{M}})$

This substitution actually takes place only on well-typed terms, so all the possible well-typed cases are those considered in the above definition. Moreover it is also a standard capture-avoid-substitution since the variables of binders  $\mathbf{def}$  and  $\mathbf{receive}$  with the same name are ignored. Every time a substitution on a variable actually takes place  $\mathit{refresh}$  is used thus fulfilling the requirement that a different refreshed version of a class or a mixin value is substituted for each occurrence of the variable  $x$ .

Thus, a general substitution can be defined using the above parametric substitution for dealing with crucial cases concerning mixin application expressions.

**Definition 5.3 (Substitution)** *The substitution  $[ / ]$  is defined as follows:*

- If  $x$  is of type  $\text{class}\langle\Sigma\rangle$  and  $C$  is a class definition of type  $\text{class}\langle\Sigma'\rangle$ , with  $\text{class}\langle\Sigma'\rangle \sqsubseteq \text{class}\langle\Sigma\rangle$ , then

$$[C/x] \stackrel{\text{def}}{=} [C/x]_{\text{Subj}(\Sigma') - \text{Subj}(\Sigma)}$$

- If  $x$  is of type  $\text{mixin}\langle\Sigma_{\text{new}}, \Sigma_{\text{red}}, \Sigma_{\text{exp}}\rangle$  and  $M$  is a mixin definition of type  $\text{mixin}\langle\Sigma'_{\text{new}}, \Sigma'_{\text{red}}, \Sigma'_{\text{exp}}\rangle$ , with  $\text{mixin}\langle\Sigma'_{\text{new}}, \Sigma'_{\text{red}}, \Sigma'_{\text{exp}}\rangle \sqsubseteq \text{mixin}\langle\Sigma_{\text{new}}, \Sigma_{\text{red}}, \Sigma_{\text{exp}}\rangle$ , then

$$[M/x] \stackrel{\text{def}}{=} [M/x]_{\text{Subj}(\Sigma'_{\text{new}}) - \text{Subj}(\Sigma_{\text{new}})}$$

- In all the other cases it is intended as the standard replacement.

Indeed, the above definition of substitution highlights a simple property: new methods added by a class or a mixin by subtyping never change their original behavior, since they will never be accidentally redefined (due to a name clash). With our solution, these new methods are hidden by applying the *refresh* during substitution. This is very similar to the “privacy via subsumption” of [33].

From the point of view of the implementation this formal treatment of “global” fresh names can be solved with static binding for the mentioned methods; the technique of using the static types of variables and the actual types of substituted mixin and class definitions may recall the approach of [19] of allowing overriding only for methods declared in the mixin’s *inheritance interface*.

## 5.2 Narrowing types

**Lemma 5.2 (Monotonicity of  $\diamond$ )** *Let  $\Gamma$ ,  $M$ ,  $\text{exp}$  and  $\text{exp}_1$  be such that*

$$\begin{array}{ll} \Gamma \vdash \text{exp} : \text{class}\langle\Sigma_b\rangle, & \\ \Gamma \vdash M \diamond \text{exp} : \text{class}\langle\Sigma_d\rangle & \text{and} \\ \Gamma \vdash \text{exp}_1 : \text{class}\langle\Sigma'_b\rangle & \text{with } \text{class}\langle\Sigma'_b\rangle \sqsubseteq \text{class}\langle\Sigma_b\rangle. \end{array}$$

*If  $\text{Subj}(\Sigma'_b - \Sigma_b) \cap \text{Subj}(\Sigma_d) = \emptyset$  then  $\Gamma \vdash (M \diamond \text{exp}_1) : \tau'$  where  $\tau' \sqsubseteq \text{class}\langle\Sigma_d\rangle$ .*

**Proof:** By typing rules, the last rule in a deduction of  $\Gamma \vdash M \diamond \text{exp} : \tau$  is a (*mixin app*) rule applied to  $\Gamma \vdash M : \text{mixin}\langle\Sigma_{\text{new}}, \Sigma_{\text{red}}, \Sigma_{\text{exp}}\rangle$  and  $\Gamma \vdash \text{exp} : \text{class}\langle\Sigma_b\rangle$ , where  $\text{class}\langle\Sigma_b\rangle$  is such that  $\Sigma_b <: (\Sigma_{\text{exp}} \cup \Sigma_{\text{red}})$  and  $\text{Subj}(\Sigma_b) \cap \text{Subj}(\Sigma_{\text{new}}) = \emptyset$ . Moreover,  $\text{class}\langle\Sigma'_b\rangle \sqsubseteq \text{class}\langle\Sigma_b\rangle$  implies  $\Sigma'_b <: \Sigma_b$ . Then we replace a derivation of  $\Gamma \vdash \text{exp}_1 : \text{class}\langle\Sigma'_b\rangle$  to  $\Gamma \vdash \text{exp} : \text{class}\langle\Sigma_b\rangle$  and the rule (*mixin app*) still applies since:

- $\Sigma'_b <: \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red})$
- $Subj(\Sigma'_b) \cap Subj(\Sigma_{new}) = \emptyset$  from the hypothesis on subjects.

Finally, the resulting type  $\tau'$  is of the shape  $\tau' \equiv \text{class}\langle \Sigma'_b \cup \Sigma_{new} \rangle$ , then  $\tau' \sqsubseteq \text{class}\langle \Sigma_d \rangle$  since  $\Sigma'_b <: \Sigma_b$ .  $\square$

In order to generalize this lemma to a generic mixin application expression of the shape  $M_n \diamond (\dots \diamond (M_1 \diamond x) \dots)$ , when we need to refer to a specific record type of a mixin expression  $M_i$  we will use the notation  $\Sigma^i$ .

**Lemma 5.3 (Narrowing Class Types)** *If*

$$\begin{array}{l} \Gamma, x : \text{class}\langle \Sigma_b \rangle \vdash M_n \diamond (\dots \diamond (M_1 \diamond x) \dots) : \tau \quad \text{and} \\ \Gamma \vdash C : \text{class}\langle \Sigma_c \rangle \quad \text{with } \text{class}\langle \Sigma_c \rangle \sqsubseteq \text{class}\langle \Sigma_b \rangle \end{array}$$

then  $\Gamma \vdash (M_n \diamond (\dots \diamond (M_1 \diamond x) \dots))[C/x] : \tau'$  where  $\tau' \sqsubseteq \tau$ .

**Proof:** By induction on the number  $n$  of applications.

**Base**  $n = 1$ . By Lemma 5.2: the hypothesis on subjects of the lemma holds since the substitution of Definition 5.3 refreshes method names belonging to  $Subj(\Sigma_c - \Sigma_x)$ .

**Ind.Step** The last rule in a deduction of  $\Gamma, x : \text{class}\langle \Sigma_b \rangle \vdash M_n \diamond (\dots \diamond (M_1 \diamond x) \dots) : \tau$  is a (*mixin app*) rule applied to  $\Gamma, x : \text{class}\langle \Sigma_b \rangle \vdash M_n : \text{mixin}\langle \Sigma_{new}^n, \Sigma_{red}^n, \Sigma_{exp}^n \rangle$  and  $\Gamma, x : \text{class}\langle \Sigma_b \rangle \vdash M_{n-1} \diamond (\dots \diamond (M_1 \diamond x) \dots) : \text{class}\langle \Sigma_{n-1} \rangle$ , where  $\Sigma_{n-1} <: (\Sigma_{exp}^n \cup \Sigma_{red}^n)$  and  $Subj(\Sigma_{n-1}) \cap Subj(\Sigma_{new}^n) = \emptyset$ .

By induction hypothesis  $\Gamma \vdash (M_{n-1} \diamond (\dots \diamond (M_1 \diamond x) \dots))[C/x] : \tau^*$  with  $\tau^* \sqsubseteq \text{class}\langle \Sigma_{n-1} \rangle$ , which implies that  $\tau^*$  is a class type,  $\tau^* \equiv \text{class}\langle \Sigma^* \rangle$ , where  $\Sigma^* <: \Sigma_{n-1}$ . Then the thesis follows from Lemma 5.2: the hypothesis on subjects, required by the lemma, holds since possible new methods added in  $\Sigma^*$  with respect to  $\Sigma_{n-1}$  are introduced by  $\Sigma_c$  and so they are renamed with fresh names by definition of substitution.  $\square$

**Lemma 5.4 (Narrowing Mixin Types)** *Let  $exp$  be a mixin application expression such that*

$$\Gamma, x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash exp : \tau$$

and let  $M$  be a mixin definition such that

$$\Gamma \vdash M : \text{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle.$$

If  $\text{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle \sqsubseteq \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$  then  $\Gamma \vdash \text{exp}[M/x] : \tau'$  where  $\tau' \sqsubseteq \tau$ .

**Proof:** By structural induction on  $\text{exp}$ . For simplicity we only consider the interesting case when  $x$  occurs in  $\text{exp}$ .

**Base**  $\text{exp} \equiv v_1 \diamond v_2$  where  $v_1 \equiv x$  and  $v_2$  is a class value.

By definition of typing rules,  $\Gamma, x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash \text{exp} : \tau$  implies that the last applied rule is a (*mixin app*) rule, that is:

$$\frac{\begin{array}{l} (1) \quad \Gamma, x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \\ (2) \quad \Gamma, x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash v_2 : \text{class}\langle \Sigma_c \rangle \\ (3) \quad \Sigma_c <: (\Sigma_{exp} \cup \Sigma_{red}) \\ (4) \quad \text{Subj}(\Sigma_c) \cap \text{Subj}(\Sigma_{new}) = \emptyset \end{array}}{\Gamma, x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash x \diamond v_2 : \text{class}\langle \Sigma_c \cup \Sigma_{new} \rangle} \text{ (mixin app)}$$

Now we replace  $\Gamma \vdash M : \text{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle$  to  $\Gamma, x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash x : \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$  and the rule (*mixin app*) still applies since:

- $\Sigma_c <: (\Sigma'_{exp} \cup \Sigma'_{red})$  follows from (3), and from  $\text{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle \sqsubseteq \text{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$  (i.e.,  $\Sigma'_{red} = \Sigma_{red}$  and  $\Sigma_{exp} <: \Sigma'_{exp}$ );
- the condition on subjects is guaranteed by Definition 5.3.

Finally,  $\tau' \equiv \text{class}\langle \Sigma_c \cup \Sigma'_{new} \rangle$  is such that  $\tau' \sqsubseteq \tau$  since  $\Sigma'_{new} <: \Sigma_{new}$ .

**Ind.Step**  $\text{exp} \equiv v \diamond \text{exp}_1$  where  $\text{exp}_1$  is a mixin application expression. We have two possible cases:

1.  $v \neq x$ . It follows from the induction hypothesis on  $\text{exp}_1[M/x]$ , using Lemma 5.2. Notice that Lemma 5.2 applies, since additional methods in  $M$  are refreshed by the substitution.
2.  $v \equiv x$ . The last applied rule is

$$\begin{array}{l}
(1) \quad \Gamma, x : \mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash x : \mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \\
(2) \quad \Gamma, x : \mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash \mathit{exp}_1 : \mathbf{class}\langle \Sigma_c \rangle \\
(3) \quad \Sigma_c <: (\Sigma_{exp} \cup \Sigma_{red}) \\
(4) \quad \mathit{Subj}(\Sigma_c) \cap \mathit{Subj}(\Sigma_{new}) = \emptyset \\
\hline
\Gamma, x : \mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash x \diamond \mathit{exp}_1 : \mathbf{class}\langle \Sigma_c \cup \Sigma_{new} \rangle \quad (\mathit{mixin\ app})
\end{array}$$

By induction hypothesis,  $\Gamma \vdash \mathit{exp}_1[M/x] : \tau^*$  with  $\tau^* \sqsubseteq \mathbf{class}\langle \Sigma_c \rangle$ , i.e.,  $\tau^* \equiv \mathbf{class}\langle \Sigma^* \rangle$  with  $\Sigma^* <: \Sigma_c$ .

Moreover,  $\mathbf{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle \sqsubseteq \mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$  implies that  $\Sigma'_{red} = \Sigma_{red}$  and  $\Sigma_{exp} <: \Sigma'_{exp}$ . Thus, the (*mixin app*) rule can be used for deducing  $\Gamma \vdash (x \diamond \mathit{exp}_1)[M/x] : \tau'$ , since

- $\Sigma^* <: \Sigma_c <: (\Sigma_{exp} \cup \Sigma_{red}) <: (\Sigma'_{exp} \cup \Sigma'_{red})$ ;
- the condition on subjects is guaranteed by the definition of substitution  $[ / ]$ . Notice that, if  $x$  occurs also in  $\mathit{exp}_1$ , then  $v \diamond \mathit{exp}_1$  is well typed only if  $\Sigma_{new} = \emptyset$ . Obviously, if  $\mathbf{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle \sqsubseteq \mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$  then  $\Sigma'_{new}$  can be such that  $\Sigma'_{new} \neq \emptyset$ . However, by definition of parametric substitution (Definition 5.2), a different refreshment of  $M$  w.r.t.  $\Sigma'_{new}$  is replaced to each occurrence of  $x$ .

□

Now we can state a general substitution property that will be needed for proving the subject reduction theorem in Section 6. For simplicity, in order to deal with  $<:$  and  $\sqsubseteq$  at the same time, we introduce the meta notation:

$$\tau_1 \preceq \tau_2 = \begin{cases} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ is a mixin or a class type} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

We firstly consider the case of replacement of values inside expressions (the generalization to replacement of expressions inside expressions would be straightforward but it is not necessary for our purpose, since only values and processes can be exchanged in the operational semantics). Then, we generalize the property to well-typed processes, that is processes having type **proc** from a suitable context  $\Gamma$  ( $\Gamma = \emptyset$  if the process is closed). Namely, an identifier  $id$  can be replaced with an argument  $A$ , being a value or a process (see Table 2), inside a process.

**Lemma 5.5** *Let  $exp$  and  $v$  be an object-oriented expression and an object-oriented value, respectively. If  $\Gamma, x : \tau_x \vdash exp : \tau$  and  $\Gamma \vdash v : \tau_1$  where  $\tau_1 \preceq \tau_x$  then  $\Gamma \vdash exp[v/x] : \tau'$  with  $\tau' \preceq \tau$ .*

**Proof:** By induction on the definition of  $exp$  (Table 1), which coincides with the induction on a deduction of  $\Gamma, x : \tau_x \vdash exp : \tau$ . The only interesting case of a mixin application expression follows from lemmas 5.3 and 5.4.  $\square$

**Lemma 5.6** *If  $\Gamma, X : \text{proc} \vdash P : \text{proc}$  and  $\Gamma \vdash Q : \text{proc}$  then  $\Gamma \vdash P[Q/X] : \text{proc}$ .*

**Proof:** Trivial, by inspection of typing rules for processes (Table 6).  $\square$

**Theorem 5.1 (Substitution Property)** *Let  $P$  be a process and let  $A$  be either an object-oriented value or a process. If*

$$\begin{array}{l} \Gamma, id : \tau \vdash P : \text{proc} \quad \text{and} \\ \Gamma \vdash A : \tau_1 \end{array}$$

*for some  $\Gamma$  and  $\tau_1 \preceq \tau$ , then  $\Gamma \vdash P[A/id] : \text{proc}$ .*

**Proof:** We distinguish two cases

- (a)  $A$  is a process. Lemma 5.6;
- (b)  $A$  is an object-oriented value. By induction on a derivation of  $\Gamma, id : \tau \vdash P : \text{proc}$ : use Lemma 5.5 for the only crucial case when  $id$  is a class or mixin variable that is replaced inside an expression in  $P$ .

$\square$

## 6 Operational Semantics

The operational semantics of MOMI is based on two sets of rules. The first one describes how object-oriented expressions reduce to values. This reduction relation, denoted by  $\twoheadrightarrow$ , is the reflexive and transitive closure of the relation  $\rightarrow$ , defined in Table 8.

The only interesting rule concerns mixin application that produces a new class containing all the methods added and redefined in the mixin and those defined in the superclass. The function *override* takes care of introducing in the new class the overridden methods, and of binding the *next* in a mixin redefined methods to the implementation provided by

the super class: such “old” methods are given a fresh name, denoted by  $m_{i'}$ . Dynamic binding is then implemented for redefined methods, and old implementations of the base class are basically hidden in the derived class, since they are given a fresh name.

**Definition 6.1** *Let  $\varrho_1$  and  $\varrho_2$  be two method sets, such that for each  $m_i : \tau_{m_i} = f_i$  belonging to  $\varrho_1$ ,  $m_i : \tau_{m_i} = f_i$  belongs to  $\varrho_2$ . The result of  $\text{override}(\varrho_1, \varrho_2)$  is the method set  $\varrho_3$  defined as follows:*

- for all  $m_i : \tau_{m_i} = f_i \in \varrho_1$ , let  $m_i = f'_i \in \varrho_2$  and let  $m_{i'}$  be a fresh method name, then  $m_{i'} = f'_i \in \varrho_3$  and  $m_i = f_i[m_{i'}/\text{next}] \in \varrho_3$
- for all  $m_i : \tau_{m_i} = f_i \in \varrho_2$  such that  $m_i \neq m_j$  for all  $m_j : \tau_{m_j} = f_j \in \varrho_1$ , then  $m_i : \tau_{m_i} = f_i \in \varrho_3$

$\frac{\text{exp} \rightarrow \{m_i : \tau_{m_i} = f_i \mid i \in I\}}{\text{exp} \leftarrow m_j \rightarrow f_j}$
$\frac{\text{exp} \rightarrow \text{class } [m_i : \tau_{m_i} = f_i \mid i \in I] \text{ end}}{\text{new exp} \rightarrow \{m_i : \tau_{m_i} = f_i \mid i \in I\}}$
$\text{exp} \rightarrow \text{class } [m_l : \tau_{m_l} = f_l \mid l \in L] \text{ end}$
$\left( \begin{array}{l} \text{mixin} \\ \text{expect}[m_i : \tau_{m_i} \mid i \in I] \\ \text{redef}[m_k : \tau_{m_k} \text{ with } f_k \mid k \in K] \\ \text{def}[m_j : \tau_{m_j} = f_j \mid j \in J] \\ \text{end} \end{array} \right) \diamond \text{exp} \rightarrow \left( \begin{array}{l} \text{class} \\ [m_j : \tau_{m_j} = f_j \mid j \in J] \cup \\ \text{override}([m_k : \tau_{m_k} = f_k \mid k \in K], [m_l : \tau_{m_l} = f_l \mid l \in L]) \\ \text{end} \end{array} \right)$

**Table 8:** Operational semantics of object-oriented expressions.

The second set of rules (presented in Table 10) 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 9) that allows the rearrangement of the syntactic structure of a term, so that reduction rules may be applied.

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

**Table 9:** Congruence laws

Reduction rules for MOMI nets are displayed in Table 10. We recall that the key idea of MOMI is that, except for the dynamic checking required during the communication, type analysis of processes is totally static and performed in each site independently. No further re-compilation and type-checking must be performed after a communication successfully took place.

To this aim, the semantics is defined on *compiled processes*  $\overline{P}$ , i.e., processes statically decorated (by the compiler) as follows:

- any `send`'s argument is decorated with its type that is derived by the type analysis algorithm (see (*send*) rule in Table 6);
- any `receive`'s argument is decorated with its type explicitly given by the programmer.

**Definition 6.2** *The compiled version of a process  $P$ , denoted by  $\overline{P}$ , is defined as follows:*

- $\overline{\mathbf{nil}} \stackrel{def}{=} \mathbf{nil}$
- $\overline{\mathbf{send}(A, \ell).P'} \stackrel{def}{=} \mathbf{send}(\overline{A}^\tau, \ell).\overline{P'}$   
*where  $\overline{A} \stackrel{def}{=} A$  if  $A$  is a value and  $\tau$  is the type assigned to  $A$  by the static type system.*
- $\overline{\mathbf{receive}(id : \tau).P'} \stackrel{def}{=} \mathbf{receive}(id^\tau).\overline{P'}$
- $\overline{P_1 \mid P_2} \stackrel{def}{=} \overline{P_1} \mid \overline{P_2}$
- $\overline{\mathbf{def } x = exp \text{ in } P'} \stackrel{def}{=} \mathbf{def } x = exp \text{ in } \overline{P'}$

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:

$$\mathit{match}(\tau_1, \tau_2) = \begin{cases} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ and } \tau_2 \text{ are mixin or class types} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

The type  $\tau_1$  of the argument  $A$  of `send` has been statically built and checked during the compilation. 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 of type  $\tau_2$ .

Rule (*def*) says that *exp* is evaluated and the resulting value is replaced to  $x$  in  $P$ . The other rules are straightforward.

$\frac{\text{match}(\tau_1, \tau_2)}{\ell_1 :: \text{send}(\overline{A}^{\tau_1}, \ell_2). \overline{P}' \parallel \ell_2 :: \text{receive}(\text{id}^{\tau_2}). \overline{Q} \succrightarrow \ell_1 :: \overline{P}' \parallel \ell_2 :: \overline{Q}[\overline{A}^{\tau_1}/\text{id}]} \text{ (comm)}$	
$\frac{\text{exp} \rightarrow v}{\ell :: \text{def } x = \text{exp in } \overline{P} \succrightarrow \ell :: \overline{P}[v/x]} \text{ (def)}$	
$\frac{N_1 \succrightarrow N'_1}{N_1 \parallel N \succrightarrow N'_1 \parallel N} \text{ (par)}$	$\frac{N \equiv N_1 \quad N_1 \succrightarrow N_2 \quad N_2 \equiv N'}{N \succrightarrow N'} \text{ (net)}$

**Table 10:** Net and process operational semantics

Preservation of types during evaluation is assumed for the concrete language of methods underlying the kernel SOOL, since it is standard for object-oriented languages. Then, this property can be easily proved for SOOL expressions:

**Lemma 6.1 (Preservation of types for Sool)** *If  $\text{exp} \rightarrow v$  and  $\Gamma \vdash \text{exp} : \tau$  then  $\Gamma \vdash v : \tau'$  where  $\tau' <: \tau$ .*

**Proof:** By structural induction on the definition of  $\rightarrow$ , Table 8. □

We observe that the type decreases from the expression to the value only because of our formal use of the *override* function in a mixin application evaluation. Namely, in order to carefully deal with the binding of *next* in the bodies of redefined methods, we have chosen to keep, in the obtained subclass, both the redefined methods and the original methods of the superclass (renamed with fresh names).

Analogously, a *progress* theorem is assumed as a standard property of the concrete underlying language and then extended, in a trivial way, to SOOL expressions (i.e., for any well-typed expression, either it is a value or it can take a step according to the evaluation rules). Then, *type safety* is commonly known as the conjunction of the progress and the type preservation theorems. However, in our mobile framework, the crucial issue we are interested in is concerned with a *global type safety* property rather than a local one. Roughly speaking, we aim at guaranteeing that a net preserves well typedness when processes and object-oriented code move among different nodes during the net evolution. To this aim, we need to formalize the notion of *well-typedness* w.r.t. a net.

**Definition 6.3** *A net  $N$  is well typed iff for any node  $\ell :: \overline{P}$  in  $N$ ,  $\Gamma \vdash P : \text{proc}$  for some  $\Gamma$ .*

Then we state the *subject reduction* property for nets:

**Theorem 6.1 (Subject Reduction)** *If  $N$  is well typed and  $N \succrightarrow N'$  then  $N'$  is well typed.*

**Proof:** By induction on  $\succrightarrow$ . First observe that if  $\Gamma \vdash a.P' : \text{proc}$  then  $\Gamma \vdash P' : \text{proc}$  (see rules in Table 6).

- (*comm*). The interesting part concerns the receiver process on  $\ell_2$ ,  $\text{receive}(id^{\tau_2}).\overline{Q}$ ; by hypothesis  $\Gamma \vdash \text{receive}(id : \tau_2).Q : \text{proc}$ , which implies that  $\Gamma, id : \tau_2 \vdash Q : \text{proc}$  (by rule (*receive*) in Table 6). Thus we use the substitution property (Theorem 5.1) and so obtain  $\Gamma \vdash Q[A/id] : \text{proc}$ .
- (*def*). Follow from Lemma 6.1 and Theorem 5.1.
- (*par*). By the induction hypothesis.
- (*net*). It is easy to verify that structural congruence (Table 9) preserves well-typedness.

□

The above theorem, together with Lemma 6.1, leads to a global safety stating that merging code received from a remote site into local code does not harm local type safety (local evaluation of well-typed object-oriented expressions cannot produce errors like “message not understood”).

Finally, 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.

## 7 A scenario for mixin mobility

We will use here a slightly simplified syntax: (*i*) we will list the methods’ parameters in between “()”; (*ii*)  $exp_1; P$  is interpreted as  $\text{def } x = exp_1 \text{ in } P$ ,  $x \notin FV(P)$ , in a call-by-value semantics; (*iii*) even though our calculus uses structural subtyping (and not nominal one), in this section we will refer to expressions related via “ $\sqsubseteq$ ” with names as shortcuts.

The example is about a client and a server, executing on two different nodes, that want to communicate, e.g., by means of a common protocol. They both use a *Socket* to this aim, however the server is willing to abstract from the implementation of such socket, by allowing the client to provide a custom implementation. This can be useful, for instance, because the client may decide to use a customized socket; in this example the client implements a socket that sends and receives compressed data (alternatively it could implement a *multicast* socket, or even a combination of the two). However, the

code sent by the client may rely on some low-level system calls, that may be different on the server's site: indeed, the two sites may run different operating systems and have different architectures. These low-level system calls are then to be provided by each site (the client's and the server's sites). The customized socket of the client is then a mixin requiring the existence of such system calls, that will be provided by two different (yet compliant) superclasses, one resident on each site. The code executed in the two nodes (`client` and `server`) is in Listing 1.

Both `ZipSocket` and `Socket` rely on a superclass that provides (at least) methods `write_to_net` and `read_from_net`. The client, in its site, completes its mixin `ZipSocket` with `NetChannel` that provides these two methods for writing data on the net, by using its operating system low-level system calls. Sending the class `ZipSocket`  $\diamond$  `NetChannel` directly to the server may be nonsense, since the server may use a different operating system (or a different version of the same operating system). Instead, only the mixin `ZipSocket` is sent to the remote server. In the server this mixin will be received as a `Socket` mixin (and this succeeds since `ZipSocket`  $\sqsubseteq$  `Socket`) and it will be completed with `NetFile`, which corresponds to the `NetChannel` of the client. The server will then use such socket independently from the particular client's implementation. Notice that the use of subtyping in the communication (instead of a simpler type equality) completely relieves the receiver (and especially its programmer) from the real complete interface of the clients' code.

Let us now consider an alternative implementation of the same scenario, in order to show other features of MOMi: suppose that on the client `ZipSocket` is written like in Listing 2 on the left. In this case the class does not rely on `write_to_net` and `read_from_net` (instead it expects the superclass to provide methods `write` and `read` that the mixin redefines), and thus it is not a subtype of `Socket` in the server. In the server, the code would be like in Listing 2 on the right. Since also `Channel` relies on a super class that provides `write` and `read`, we have that `ZipSocket`  $\sqsubseteq$  `Channel`. So the server receives a `ZipSocket` (as a `Channel`) that it completes with `Socket` completed, in turn, with `NetFile`.

Finally, as hinted in Section 3.1, other implementations of such a socket can be created, simply by using more than one mixin, such as `UUEncode`, `Encrypt`, and so on.

## 8 Conclusions and Related Work

In the literature, there are several proposals of combining objects with processes and/or mobile agents. *Obliq* [12] is a lexically-scoped language providing distributed object-

```

client:: let ZipSocket =
  mixin
    def zip = ...
    def unzip = ...
    def write = write_to_net(zip(data))
    def read = unzip(read_from_net())
    expect write_to_net
    expect read_from_net
  end in
let NetChannel =
  class
    send_data =
      // <send through the net>
    receive_data =
      // <receive from the net>
    write_to_net = send_data(data)
    read_from_net = receive_data()
  end in
let channel = new
  (ZipSocket ◊ NetChannel) in
  (
    send( ZipSocket, server ).
    ( channel<=>write("hello") ;
      channel<=>read() )
  )
)

server:: let Socket =
  mixin
    def write = write_to_net(data)
    def read = read_from_net()
    expect write_to_net
    expect read_from_net
  end in
let NetFile =
  class
    write_to_net =
      // <send through the net>
    read_from_net =
      // <receive from the net>
  end in
  (
    receive( sock : Socket ).
    let client_channel = new
      (sock ◊ NetFile) in
      (
        client_channel<=>read() ;
        client_channel<=>write("welcome")
      )
  )
)

```

**Listing 1:** Example code for client and server communication.

<pre> ZipSocket =   mixin   def zip = ...   def unzip = ...   redef write = next(zip(data))   redef read = unzip(next()) end </pre>	<pre> Channel =   mixin   redef write = next(data)   redef read = next() end ... receive( chan : Channel ). let client_channel =   new (chan ◊ (Socket ◊ NetFile)) in   (     client_channel←read() ;     client_channel←write("welcome")   ) </pre>
-------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Listing 2:** An alternative implementation.

oriented computation. Mobile code maintains network references and provides transparent access to remote resources. In [11], a general model for integrating object-oriented features in calculi of mobile agents is presented: agents are extended with method definitions and constructs for remote method invocations. Other works, such as, e.g., [16, 32, 24, 21] are concerned in merging concurrency and object orientation and do not deal explicitly with mobile distributed code. In our calculus no remote method call functionality is considered, and, instead of formalizing remote procedure calls (like most of the above mentioned approaches), MOMI provides the introduction of safe and scalable distribution of object-oriented code, in a calculus where communication and coordination facilities are already provided.

The crucial feature of MOMI is that classes and mixins are themselves mobile code, i.e., code that is dynamically downloaded from a remote source. This implies that, at application time, the actual implementation of mixins and classes may differ from their expected (static) interface (obviously the two must be compliant by subtyping, as described in Section 4.3). This highlights the main difference between our approach and other mixin-based languages in the literature (see, e.g., [19, 2]): in most mixin-based languages a mixin can be applied to several base classes (and this shows the “dynamic” inheritance nature of mixins) but when the application takes place in a specific part of a program, both the code of the mixin and the one of the classes are available for the compiler. This does not hold in MOMI since mixin application can act also on mixin and class variables. So the tasks

of generating a new class, when a mixin is applied to a class, and of creating an object from a class (performing all the right bindings) must be done at run-time in MOMI, when the class and mixin variables have been actually instantiated with effective code (possibly downloaded from the network). This is similar to what it happens in calculi where classes and mixins are “first-class citizens”, i.e., they can be passed as parameters to functions (see, for example, [9]). It is important to notice, though, that in such calculi the matching between actual and formal parameters is always based on equality (at least at the best of our knowledge), because the burden of checking extra constraints at parameter-passing time, to avoid inheritance/subtyping conflicts, is not worthwhile in a sequential scenario. In a mobile distributed scenario, instead, where flexibility is a must because the nature of the downloaded code cannot always be estimated a priori, the use of such extra subtyping constraints at communication time is a small price to pay, especially since this allows to combine local and foreign code without any recompilation.

With regard to work-in-progress, we are designing a mixin-oriented version of KLAIM (in particular, a prototype implementation of KLAIM extended with MOMI’s features is presented in [5]). Since KLAIM is based on the Linda concept of *tuple space* [23] and asynchronous communication, the integration of MOMI in KLAIM also shows that the migration to an asynchronous version of the distributed calculus with mixins is quite smooth. Moreover, the type system presented in this paper can be modified in order to refine *proc* types, so that finer types can be assigned to processes, e.g., according to the capability-based type system for access control developed for KLAIM [15].

We are also investigating how to extend MOMI and its type system with *incomplete objects* (i.e., instantiated from mixins, instead of classes), so that they can be completed by using a *method addition* operation, possibly in other sites. The main issue here is to type incomplete objects in such way this type information would allow them to be safely completed on remote sites (we thus plan to model a well-behaved object-based calculus internal to the mixin-based one).

From a more technical point of view, we are extending our width subtyping for record types with depth subtyping. This extension allows to achieve an even more powerful mixin and class subtyping, and so increases the communication flexibility in a substantial way. However, this powerful subtyping raises many technical difficulties and requires further static analysis. A first treatment of this problem is presented in [7].

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 [29] 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 [28, 4] for some techniques for avoiding these problems in concurrent object-oriented languages.

## 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] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.
- [4] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. In B. Magnusson, editor, *Proc. of ECOOP02*, volume 2374 of LNCS, pages 415–440. Springer, 2002.
- [5] L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. forthcoming.
- [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] L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Classes and Mixins. In *Proc. of Int. Workshops on Foundations of Object-Oriented Languages, FOOL 10*, 2003. To appear.
- [8] L. Bettini, M. Loreti, and B. Venneri. On Multiple Inheritance in Java. In *Proc. of TOOLS EASTERN EUROPE, Emerging Technologies, Emerging Markets*, 2002. To appear.

- [9] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In R. Guerraoui, editor, *Proceedings ECOOP'99*, number 1628 in LNCS, pages 43–66. Springer-Verlag, 1999.
- [10] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.
- [11] M. Bugliesi and G. Castagna. Mobile Objects. In *Proc. of FOOL*, 2000.
- [12] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
- [13] A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R. Taylor, editor, *Proc. of the 19th Int. Conf. on Software Engineering (ICSE '97)*, pages 22–33. ACM Press, 1997.
- [14] 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.
- [15] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
- [16] 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.
- [17] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ICFP '98*, pages 94–104, 1998.
- [18] 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.
- [19] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
- [20] C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of LNCS, pages 406–421. Springer-Verlag, 1996.

- [21] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the Join Calculus. In S. Kapoor and S. Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS2000)*, volume 1974 of *LNCS*, pages 397–408. Springer, 2000.
- [22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [23] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
- [24] 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.
- [25] C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report 19887, IBM Research Division, 1994.
- [26] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
- [27] F. Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS workshop on Analysis and Verification of Multiple - Agent Languages*, number 1192 in *LNCS*. Springer-Verlag, 1996.
- [28] 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.
- [29] 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.
- [30] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [31] Object Management Group. Corba: Architecture and specification. <http://www.omg.org>, 1998.

- [32] 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.
- [33] J. Riecke and C. Stone. Privacy via Subsumption. *Information and Computation*, (172):2–28, 2002. 3rd special issue of Theory and Practice of Object-Oriented Systems (TAPOS).
- [34] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP '98*, pages 550–570, 1998.
- [35] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes IRISA.
- [36] M. Van Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
- [37] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. OOPSLA '96*, pages 359–369, 1996.
- [38] J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.