SUBTYPING-INHERITANCE CONFLICTS: THE MOBILE MIXIN CASE*

Lorenzo Bettini[1] Viviana Bono[2] Betti Venneri[1]
[1] *Dipartimento di Sistemi e Informatica, Università di Firenze*
[2] *Dipartimento di Informatica, Università di Torino*
[1] {bettini,venneri}@dsi.unifi.it, 2bono@di.unito.it

Abstract In sequential class- and mixin-based settings, subtyping is essentially a relation on objects: no subtype relation is defined on classes and mixins, otherwise there would be conflicts with the inheritance mechanism, creating type un-safety. Nevertheless, a width-depth subtyping relation on class and mixin types is useful in the realm of mobile and distributed processes, where object-oriented code may be exchanged among the sites of a net. In our proposal, classes and mixins become "first-class citizens" at communication time, and communication is ruled by a type-safe width-depth subtyping relation.

1. Introduction

In sequential class-based settings, and similarly in sequential mixin-based settings, subtyping is essentially a relation on objects. Either no subtype relation (as in [9]), or no non-trivial subtype relation is defined on classes and mixins, otherwise there would be conflicts with the inheritance mechanism (see [11], Chapter 5.3). Our goal is to study a subtyping relation extended to classes and mixins in the realm of mobile and distributed processes, where object-oriented code can be exchanged among the sites of a network. Classes and mixins become "first-class citizens" at communication time, and communication is ruled by the subtyping relation.

In [5], we introduced MOMI (Mobile Mixins), a core coordination calculus for mobile processes that exchange mixin-based object-oriented code. The leading idea of MOMI is that the intrinsic "incompleteness" of mixins, which are incomplete classes parameterized over a superclass [10, 2, 17], makes mixin-based inheritance more suited than classical class-based inheritance to model mobile code. The most important feature of MOMI's typing is a *subtype* relation that guarantees safe, yet flexible, code communication. We assume that the code that is communicated has been successfully compiled, and that it travels together with its static type. When the

*This work has been partially supported by EU within the FET - Global Computing initiative, project AGILE IST-2001-32747, project DART 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.

code is received on a site (whose code also has been successfully compiled), it is accepted only if its type is subtyping-compliant with respect to the one expected. If the code is accepted, it can interact with the local code in a safe way (i.e., with no run-time errors), without any further type checking of the whole code.

The proposed subtype relation on classes and mixins is far from straightforward. In fact, it is well known that subtyping and inheritance do not interact well: problems mirroring the "width subtyping versus addition" and "depth subtyping versus override" conflicts in the object-based setting [1, 15, 8, 20] also arise in our setting. Our contribution is to solve comprehensively both conflicts in the setting of mobile mixin-based code, enforcing a correct substitution property. The effort of defining a class-mixin subtype relation and the related dynamic checking at communication time is worthwhile in a distributed setting, where it is not predictable how mobile code will be used when transmitted to different remote contexts, and, symmetrically, a certain site must allow some controlled flexibility in accepting foreign code.

2. MOMI: Mobile Mixin Calculus

			v	::=	$\{m_i = f_i\ ^{i\in I}\}$
				|	x
exp	::=	v		|	class $[m_i = f_i\ ^{i\in I}]$ end
	|	new exp			mixin
	|	$exp \Leftarrow m$			expect$[m_i : \tau_{m_i}\ ^{i\in I}]$
	|	$v \diamond exp$		|	redef$[m_k : \tau_{m_k}$ as τ'_{m_k} with $f_k\ ^{k\in K}]$
					def$[m_j = f_j\ ^{j\in J}]$
					end

Table 1. **Syntax of SOOL.**

The calculus MOMI has an object-oriented mixin-based component, and a coordination component including representative features for distribution, communication and mobility of processes and code. MOMI supports mixin-based class hierarchies via *mixin definition* and *mixin application.* 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.

SOOL expressions offer object instantiation, method call and *mixin application*; $\diamond$ denotes the mixin application operator and it associates to the right. A SOOL value, to which an expression reduces, is either an object, which is essentially a (recursive) record $\{m_i = f_i\ ^{i\in I}\}$, or a class definition, or a mixin definition, where $[m_i = f_i\ ^{i\in I}]$ denotes a sequence of method definitions, and $[m_k : \tau_{m_k}$ as τ'_{m_k} with $f_k\ ^{k\in K}]$ denotes a sequence of method re-definitions, where τ_{m_k} is the type of the original method m_k in the superclass and τ'_{m_k} is the type of the redefining method body f_k of m_k in the mixin. 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.

Another assumption we make is that methods do not accept/return classes and mixins as parameters/results, in order to keep the algorithm of Section 6 technically simpler.

A mixin is essentially an abstract class that is parameterized over a (super)class. Each mixin consists of three parts: (*i*) methods defined in the mixin; *(ii) expected methods,* that must be provided by the superclass; (*iii*) *redefined methods,* 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	::=	**nil**	(null process)
	|	$a.P$	(action prefixing)
	|	$P_1 \mid P_2$	(parallel comp.)
	|	X	(process variable)
	|	def $x = exp$ in P	(def)
a	::=	send(A, ℓ)	(send)
	|	receive$(id : \tau)$	(receive)
A	::=	$v \mid P$	(send's arg.)
id	::=	$x \mid X$	(receive's arg.)
N	::=	$\ell :: P$	(node)
	|	$N_1 \parallel N_2$	(net composition)

Table 2. MOMI syntax.

MOMI's coordination component is similar to CCS [18] but also inspired by KLAIM [14], 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, **send**(A, ℓ) 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 receive. The argument of receive, *id,* ranges over x (a variable of SOOL) and X (a process variable).

3. Typing

The set $\mathcal{T}$ of types for SOOL is defined as follows:

$$\tau ::= \Sigma \mid \textbf{class}\langle\Sigma\rangle \mid \textbf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle \qquad \Sigma ::= \{m_i : \tau_{m_i}{}^{i \in I}\}$$

Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i}{}^{i \in I}\}$. If $m_i : \tau_{m_i} \in \Sigma$ we say that the *subject* m_i *occurs in* Σ. $Subj(\Sigma)$ is the set of the subjects of Σ and $Meth(\Sigma)$ is the set of all the method names occurring in Σ (e.g., if $\Sigma = \{m : \{n : \tau\}\}$, then $Subj(\Sigma) = \{m\}$ and $Meth(\Sigma) = \{m, n\}$). 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 denote this type derivability with $\vdash$, and $\vdash$-**statements** are used as assumptions in typing values. 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.

Class types **class**$\langle\Sigma\rangle$ and mixin types **mixin**$\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle$ are formed over record types. A class type collects the types of its methods $\{m_i : \tau_{m_i}{}^{i \in I}\}$. The typing rule for mixin values is in Table 3 (typing rules for classes and other values are straightforward and therefore omitted). A mixin type encodes the following information, Σ_{new}, Σ_{red} are the types of the mixin methods (new and redefining, 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. In Σ_{exp} there are the types of the methods that are not redefined by the mixin but expected to be supported by the superclass. In Σ_{old} there are the types assumed for the superclass bodies of the methods redefined by the mixin. We

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

Table 3. Typing rule for mixin values.

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

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

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

Table 4. Typing rules for SOOL expressions.

refer to both sets of types Σ_{exp} and Σ_{old} 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 typing rules for SOOL expressions are in Table 4. The crucial 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, 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 [19]:

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

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, thus $\Sigma_1 \cup \Sigma_2$ 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 needed 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)* $\Sigma_{red} <: \Sigma_b/\Sigma_{red}$ checks that the types of the methods redefined by the mixin (Σ_{red}) are subtypes of the superclass methods with the same name; *(iii)* $Meth(\Sigma_b) \cap Meth(\Sigma_{new}) = \emptyset$ guarantees that no name clash takes place during the mixin application. Intuitively, the above constraints insure that all the actual method bodies of the newly created sub-class are at least as "good" as expected. The resulting class, of type $\mathbf{class}\langle\Sigma_d\rangle$, 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.

Typing rules for processes are defined in Table 5. At this stage, we are not interested in typing processes in detail, therefore we will simply assign to a well-typed process the constant type proc, which means that the object-oriented code the process may contain is well typed. The set $\mathcal{T}$ of types is extended to $\mathcal{T}^* = \mathcal{T} \cup \{\mathsf{proc}\}$. Typing environments are extended with assertions $id : \tau$, where *id* ranges over *x* and *X* and τ ranges over $\mathcal{T}^*$.

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

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

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

Table 5. **Typing rules for processes.**

The rules are auto-explicative. Notice that if a process *P* has type proc, then all object-oriented expressions occurring in *P* are typed. Finally, we 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 $\mathbf{send}(A, \ell)$ occurring in *P*, the free variables of *A* are bound by an outer def or by an outer receive. This implies that the exchanged code is closed when a send is executed. Notice also that all typing rules characterizing our calculus are in an algorithmic form.

4. Subtyping on Classes and Mixins

The key point of our approach is the introduction of a subtyping relation, $\sqsubseteq$, on class and mixin types. It is of paramount importance to notice that $\sqsubseteq$ is never used in the (local) static type inference. Only during communication the actual parameter type will be matched against the formal parameter type by $\sqsubseteq$ in order to synchronize a send action with a receive one. Therefore, in our mobile scenario, classes and mixins get a polymorphic and higher-order nature only during the mobile code exchange via $\sqsubseteq$. The subtyping relation $\sqsubseteq$ is defined in Table 6. The rule $(\sqsubseteq\ class)$ is naturally

induced by the depth-and-width subtyping on record types. The rule ($\sqsubseteq$ *mixin*): (*i*') allows the subtype to define more new methods; (*ii*) requires the subtype to override the same methods; *(iii)* allows a subtype to require fewer expected methods.

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

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

Table 6. **Subtype on class and mixin types.**

The communication mechanism is implemented by annotating the send's argument with its type during the static type analysis. Therefore, it is possible to replace the formal parameter inside a process *P* with the sent code if its type is subtyping-compliant with the expected one, without requiring any further type checking. To guarantee this, we must prove that our type system enjoys a property of *substitutivity,* i.e., well-typedness is preserved under substitution by $\sqsubseteq$. Concerning this issue, width and depth subtyping raises two orthogonal problems that mirror their counterparts in the object-based setting [1, 15, 8, 20]. We solve those problems, and prove a global substitutivity property, in the sequel.

5. Width Subtyping vs Method Addition: Refreshing

Accidental overrides can occur when replacing at run-time *M* or *C* with M_1 and C_1 of smaller types in a mixin application $M \diamond C$, because of names of new methods possibly added by M_1 or C_1. This is related to the "width subtyping versus method addition" problem (well-known in the object-based setting, see for instance [15]), that in our case boils down to a careful management of such *dynamic name clashes.* Thus, we define a suitable capture-avoid-substitution, denoted with [ ], requiring possible renaming of methods with fresh names.

DEFINITION 1 (SUBSTITUTION BY REFRESH) *If x is a class variable of type* $\mathsf{class}\langle\Sigma\rangle$ *and C is a class value of type* $\mathsf{class}\langle\Sigma'\rangle$ *such that* $\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$, *then* $[C/x]$ *denotes the replacement of* C' *to x, where* C' *is obtained from C by renaming all methods belonging to* $Meth(\Sigma') - Meth(\Sigma)$ *with fresh names. If x is a mixin variable of type* $\mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle$ *and M is a mixin value of type* $\mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}, \Sigma'_{old}\rangle$ *such that* $\mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}, \Sigma'_{old}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle$, *then* $[M/x]$ *denotes the replacement of* M' *to x, where* M' *is obtained from M by renaming all methods belonging to* $Meth(\Sigma'_{new}) - Meth(\Sigma_{new})$ *with fresh names. In all remaining cases, substitution is intended as a standard replacement.*

With our solution, new methods added by a class or a mixin value during substitution are hidden by renaming, for each occurrence of the variable to be replaced (this is similar to the "privacy via subsumption" of [20]). Notice that we only rename methods that do not appear in the type of the variable *x.* This constraint ensures that the sub-

typing relation is preserved by the refreshed version. This basic property is necessary for proving that the substitution is type-safe (Theorem 11).

PROPERTY 1 (REFRESHING PRESERVES SUBTYPING) *Let* v *be a class value or a mixin value. If* $\Gamma \vdash v : \tau_1$ *and* $\Gamma \vdash x : \tau_2$ *with* $\tau_1 \sqsubseteq \tau_2$, *then* $\Gamma \vdash [v/x] : \tau_1'$ *with* $\tau_1' \sqsubseteq \tau_2$.

From the point of view of the implementation, the above treatment of "global" fresh names can be solved with static binding for the mentioned methods. The technique of using the static types of the variables and the actual types of the substituted class or mixin definitions may recall the approach of [17] of allowing overriding, i.e., dynamic binding, only for methods declared in the mixin's *inheritance interface.*

6. Depth Subtyping vs Override: Annotating Processes

Let P be a closed process to be compiled. While reconstructing the derivation of $\emptyset \vdash P : \mathsf{proc}$ (this derivation is unique, see the typing rules), it is easy to decorate any send argument occurring in P with its type. For instance, $\mathsf{def}\ x = exp\ \mathsf{in}\ \mathsf{send}(x, \ell)$ has type proc, and its compiled version is $\mathsf{def}\ x = exp\ \mathsf{in}\ \mathsf{send}(x^{\tau_1}, \ell)$ if *exp* has type τ_1.

However, this type information is not sufficient for dynamic matching, since the presence of depth subtyping conflicts with the overriding inheritance mechanism. First, we present an example (which is directly adapted from the classical one related to the object-based case of [1]). Let us consider the following expression:

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

where M is a mixin redefining n with body -3. Now, receive could accept as an actual parameter a fully-fledged class $C : \mathsf{class}(\{m : \mathsf{int}, n : \mathsf{posint}\})$, where the actual body of m is $\mathsf{log}(self \Leftarrow n)$ (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 $x : \mathsf{class}(\{m : \tau\})$ 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 substitute dynamically to such x any received class $C : \mathsf{class}(\{m : \tau_2\})$, with $\mathsf{class}(\{m : \tau_2\}) \sqsubseteq \mathsf{class}(\{m : \tau\})$, 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 same problem can arise when replacing a mixin value M to a mixin variable *x, e.g.,* in a mixin application of the shape $M_1 \diamond (x \diamond C)$. In fact, some new method m might be of type τ_2 in the (Σ_{new} of the) type of M (see ($\sqsubseteq$ *mixin*) rule in Table 6), while it is of type τ in x and redefined by M_1 as $m : \tau_1$, with $\tau_2 <: \tau$ and $\tau_1 <: \tau$. Again, $M_1 \diamond (M \diamond C)$ is well typed if and only if $\tau_1 <: \tau_2 <: \tau$.

As a consequence, the formal parameter of a receive, if it is of type "class" or "mixin", 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" or "mixin" variable can appear inside a chain of mixin applications, and this may give rise to several constraints concerning several methods. Any receive's argument of type "class" or "mixin" will be then annotated with both its type and a type assertion $\mathfrak{A}$, which will contain no lower bound if the parameter does not participate in any mixin application.

The algorithm presented in Tables 7 and 8 performs all the above type annotations while checking well-typedness of processes. We remark that the preliminary version of this algorithm sketched in [6] was a restriction of the present one, since depth subtyping was only considered on classes (not on mixins).

DEFINITION 2 *A* type assertion $\mathfrak{A}$ *is a property of the* ***shape*** $\mathfrak{A} = \mathsf{inf}(x:\tau):\Sigma'$, *where* τ *is either a class or mixin type and:*

- Σ' *can be empty* ($\Sigma' = \emptyset$);
- *if* $\tau \equiv \mathsf{class}\langle\Sigma\rangle$,
  - $Subj(\Sigma') \subseteq Subj(\Sigma)$;
  - *if* $m:\tau' \in \Sigma'$ *then* $m:\tau \in \Sigma$ *with* $\tau' <: \tau$, *for some* τ;
- *if* $\tau \equiv \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle$,
  - $Subj(\Sigma') \subseteq Subj(\Sigma_{new})$;
  - *if* $m:\tau' \in \Sigma'$ *then* $m:\tau \in \Sigma_{new}$ *with* $\tau' <: \tau$, *for some* τ.

Informally speaking, Σ' acts as an "inf" for Σ (resp. Σ_{new}), since it contains lower bounds for some (possibly none) of the types associated to methods in Σ (resp. Σ_{new}). We define $label(\mathfrak{A})$ as follows: $label(\mathsf{inf}(x:\tau):\Sigma') = x$. We define $rectype(\mathfrak{A})$ as follows: $rectype(\mathsf{inf}(x:\mathsf{class}\langle\Sigma\rangle):\Sigma') = \Sigma$; and $rectype(\mathsf{inf}(x:\mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle):\Sigma') = \Sigma_{new}$.

DEFINITION 3 *Let* τ *be a class or mixin type and* $\mathfrak{A}$ *be a type assertion,* $\mathfrak{A} = \mathsf{inf}(x:\tau'):\Sigma'$. *We say that* τ satisfies $\mathfrak{A}$, *denoted by* $\tau \models \mathfrak{A}$, *if and only if*

- $\tau \sqsubseteq \tau'$;
- $\tau \equiv \mathsf{class}\langle\Sigma\rangle \Rightarrow \Sigma/\Sigma' :> \Sigma'$;
- $\tau \equiv \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle \Rightarrow \Sigma_{new}/\Sigma' :> \Sigma'$.

In other words, $\mathsf{class}\langle\Sigma\rangle \models \mathsf{inf}(x:\Sigma_1):\Sigma_2$ 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 holds trivially. For instance, the type $\tau = \mathsf{class}(\{m_1:\tau_1, m_2:\tau_2, m_3:\tau_3\})$ satisfies the assertion $\mathfrak{A} = \mathsf{inf}(x:\mathsf{class}(\{m_1:\tau_1^b, m_2:\tau_2^b\})):\{m_1:\tau_1^{red}, m_2:\tau_2^{red}\}$, provided that $\tau_1 <: \tau_1^b$ and $\tau_2 <: \tau_2^b$, and that $\tau_1^{red} <: \tau_1$ and $\tau_2^{red} <: \tau_2$. The clause on mixins is analogous, on the component Σ_{new}. We can collect assertions for several distinct variables, therefore obtaining a *type effect.*

DEFINITION 4 *A* type effect $\mathcal{E}$ *is a set, possibly empty, of type assertions* $\mathcal{E} = \{\mathfrak{A}_1, \ldots, \mathfrak{A}_n\}$, *where* $label(\mathfrak{A}_i) \neq label(\mathfrak{A}_j)$, $1 \leq i, j \leq n$, *for* $i \neq j$.

An *annotated process,* denoted by $\overline{P}$, is a process decorated by adding: (*i*) types to the arguments of its send's; *(ii)* and types and type assertions to the arguments of its receive's. 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 the derived type effect. Then, we define $Ann(\Gamma, P)$ that returns $\langle \overline{P}, \mathbf{proc}, \mathcal{E} \rangle$: $\overline{P}$ is the annotated version of *P,* proc means that *P* is well typed in Γ and $\mathcal{E}$ is a type effect. In both cases, the algorithm fails if the expression or the process are not typable, but here we do not handle failures explicitly.

$Ann(\Gamma, v)$:
  let $\tau = \Gamma(v)$ in
  if v is a variable and τ is class or mixin type then
    $\langle v, \tau, \{\mathbf{inf}(v:\tau) : \emptyset\} \rangle$
  else
    $\langle v, \tau, \emptyset \rangle$

$Ann(\Gamma, \mathbf{new}\ exp)$:
  let $\langle exp, \mathbf{class}\langle \Sigma \rangle, \mathcal{E} \rangle = Ann(\Gamma, exp)$ in
    $\langle \mathbf{new}\ exp, \Sigma, \mathcal{E} \rangle$

$Ann(\Gamma, exp \Leftarrow m)$:
  let $\langle exp, \mathbf{class}\langle \{\ldots m:\tau \ldots\} \rangle, \mathcal{E} \rangle = Ann(\Gamma, exp)$ in
    $\langle exp \Leftarrow m, \tau, \mathcal{E} \rangle$

$Ann(\Gamma, v \diamond exp)$:
  let $\langle v, \mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle, \{\mathfrak{A}\} \rangle = Ann(\Gamma, v)$ in
  let $\langle exp, \mathbf{class}\langle \Sigma_b \rangle, \mathcal{E} \rangle = Ann(\Gamma, exp)$ in
  let $\Sigma_{rest} = (\Sigma_b - (\Sigma_b/\Sigma_{exp} \cup \Sigma_b/\Sigma_{old}))$ in
  let $\Sigma_d = \Sigma_b/\Sigma_{exp} \cup \Sigma_{new} \cup \Sigma_{red} \cup \Sigma_{rest}$ in
    $\langle v \diamond exp, \mathbf{class}\langle \Sigma_d \rangle, update(\mathcal{E}, \Sigma_{red}) \cup \{\mathfrak{A}\} \rangle$

Table 7. **The annotation algorithm for expressions.**

The algorithm *Ann* on expressions is in Table 7 and it is defined inductively on the structure of expressions. For simplicity, we use the notation $\Gamma(v)$ to denote the type τ such that $\Gamma \vdash v:\tau$ for any value v, not only for the variables occurring in Γ. Type assertions are neither generated nor modified by class and mixin definitions. The only values affecting them are variables of class or mixin types. When the algorithm is called on a variable x of class or mixin type τ, it creates a new assertion $\mathbf{inf}(x:\tau) : \emptyset$ where the lower bound for τ is temporarily empty. This lower bound will be defined by examining the possible occurrences of x inside mixin applications present in the expression. Notice that the final type effect collected by the algorithm can consist of several type assertions, since different free variables of mixin and class types can occur inside the same expression.

Cases of new *exp* and $exp \Leftarrow m$ are simple. The only interesting case concerns mixin application expressions of the shape $x \diamond exp$. In this case, $Ann(\Gamma, x \diamond exp)$ recursively calls *Ann* on x and *exp,* therefore obtaining a type assertion $\mathfrak{A}$ and a type effect $\mathcal{E}$, respectively. Let x be of type $\mathbf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle$. Now $\mathcal{E}$ must be firstly updated by using Σ_{red}, and then the resulting type effect must be extended with the new type assertion $\mathfrak{A}$. The first operation is performed by the function *update,* which is formally defined in Definition 5. The function $update(\mathcal{E}, \Sigma_{red})$ enriches $\mathcal{E}$ with lower bounds associated to any method $m:\tau$ belonging to Σ_{red} in the following way: *(i)* for all assertions of $\mathcal{E}$ of the shape $\mathbf{inf}(y:\mathbf{class}\langle \Sigma \rangle) : \Sigma_1$, where $m \in Subj(\Sigma)$, if m has

no lower bound in Σ_1, then the new lower bound $m:\tau$ is added to Σ_1; (*ii*) analogously, for assertions $\mathbf{inf}(y:\mathbf{mixin}\langle\Sigma'_{new},\Sigma'_{red},\Sigma'_{exp},\Sigma'_{old}\rangle):\Sigma'_1$, where $m \in Subj(\Sigma'_{new})$.

Thus, $update(\mathcal{E},\Sigma_{red})$ defines the lower bound associated to the method name m only if a lower bound for m had not already been defined; this guarantees that the greater lower bound for any redefined method is stored in the assertion. Finally, the assertion $\mathfrak{A}$, generated by the mixin value x, is added to the result of *update*. Notice that all this is based on the fact that mixin applications are well typed, thus, if x occurs twice in the same mixin application expression, its Σ_{new} must be empty and therefore $update(\mathcal{E},\Sigma_{red}) \cup \{\mathfrak{A}\}$ is well defined.

DEFINITION 5 *Given an effect* $\mathcal{E}$ *and a record type* Σ', $update(\mathcal{E},\Sigma')$ *is the type effect* $\mathcal{E}'$ *defined as follows:*
for each assertion $\mathbf{inf}(x:\tau):\Sigma_1 \in \mathcal{E}$, *let* $\Sigma = rectype(\mathbf{inf}(x:\tau):\Sigma_1)$;

1 *if* $Subj(\Sigma) \cap Subj(\Sigma') \neq \emptyset$ *then* $\mathbf{inf}(x:\tau):\Sigma_1 \cup \Sigma_2 \in \mathcal{E}'$, *where* $\Sigma_2 = \{m_i:\tau_i \mid m_i:\tau_i \in \Sigma' \wedge m_i \notin Subj(\Sigma_1)\}$;
2 *otherwise,* $\mathbf{inf}(x:\tau):\Sigma_1 \in \mathcal{E}'$.

The algorithm *Ann* for processes is in Table 8 and is defined inductively on the structure of processes or, equivalently, on typing rules for processes. The resulting $\mathcal{E}$ will contain type assertions for all of the variables occurring in the mixin application subterms of the process. Notice that a free variable can have different occurrences in a process P, in particular, it can occur in different sub-processes, giving raise to different type effects, one for each sub-process. Thus, when *Ann* is called on the process $P_1 \mid P_2$ **(on def** $x = exp$ **in** P**)** type effects obtained by recursive calls on P_1 and P_2 (on *exp* and P) must be merged according to the Definition 7 of *merge*. Namely, if P_1 and P_2 (*exp* and P) produce two distinct type assertions corresponding to the same variable, then the maximum lower bound for every method is collected (which always exists by well-typedness and Definitions 6 and 7).

$$
\begin{array}{l}
Ann(\Gamma,X): \\
\quad \text{if proc} = \Gamma(X) \text{ then} \\
\quad \langle X, \text{proc}, \emptyset\rangle
\end{array}
\qquad
\begin{array}{l}
Ann(\Gamma, \text{send}(A,\ell).P): \\
\quad \text{let } \langle \overline{A}, \tau, \mathcal{E}\rangle = Ann(\Gamma, A) \text{ in} \\
\quad \text{let } \langle \overline{P}, \text{proc}, \mathcal{E}'\rangle = Ann(\Gamma, P) \text{ in} \\
\quad \langle \text{send}(\overline{A}^{\tau},\ell).\overline{P}, \text{proc}, merge(\mathcal{E},\mathcal{E}')\rangle
\end{array}
$$

$$
\begin{array}{l}
Ann(\Gamma, \text{receive}(id:\tau).P): \\
\quad \text{let } \langle \overline{P}, \text{proc}, \mathcal{E}\rangle = Ann(\Gamma \cup \{id:\tau\}, P) \text{ in} \\
\quad \langle \text{receive}(id^{\tau\mid(\mathcal{E}\downarrow id)}).\overline{P}, \text{proc}, \mathcal{E} - (\mathcal{E}\downarrow id)\rangle
\end{array}
$$

$$
\begin{array}{l}
Ann(\Gamma, \text{def } x = exp \text{ in } P): \\
\quad \text{let } \langle exp, \tau, \mathcal{E}\rangle = Ann(\Gamma, exp) \text{ in} \\
\quad \text{let } \langle \overline{P}, \text{proc}, \mathcal{E}'\rangle = Ann(\Gamma \cup \{x:\tau\}, P) \text{ in} \\
\quad \langle \text{def } x = exp \text{ in } \overline{P}, \text{proc}, merge(\mathcal{E},\mathcal{E}')\rangle
\end{array}
\qquad
\begin{array}{l}
Ann(\Gamma, P_1 \mid P_2): \\
\quad \text{let } \langle \overline{P_1}, \text{proc}, \mathcal{E}_1\rangle = Ann(\Gamma, P_1) \text{ in} \\
\quad \text{let } \langle \overline{P_2}, \text{proc}, \mathcal{E}_2\rangle = Ann(\Gamma, P_2) \text{ in} \\
\quad \langle \overline{P_1} \mid \overline{P_2}, \text{proc}, merge(\mathcal{E}_1,\mathcal{E}_2)\rangle
\end{array}
$$

Table 8. **The annotation algorithm for processes.**

When *Ann* is called on a $\mathbf{send}(A,\ell).P$, the argument A is annotated with its type, while the effect generated by A is merged with the one collected when annotating the continuation P. When *Ann* is called on $\mathbf{receive}(id:\tau).P$, the variable id is annotated

with its type τ and with the assertion on the subject *id* that is possibly generated during the recursive call of *Ann* on the continuation P ($\mathcal{E} \downarrow id$ is $\mathbf{inf}(id:\tau):\Sigma$ if $\mathbf{inf}(id:\tau):\Sigma \in \mathcal{E}$, and $\emptyset$ otherwise). Since receive is a binder for *id*, it makes sense to discard the assertions for *id* from the type effect ($\mathcal{E} - (\mathcal{E} \downarrow id)$) after annotating the receive, thus the final $\mathcal{E}$ is empty when starting from a closed P.

In the following we define formally the function *merge* that takes two type effects and builds a new type effect.

DEFINITION 6 *Given the types* τ, τ_1 *and* τ_2 *we define*

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

DEFINITION 7 *Given two type effects* $\mathcal{E}_1$ *and* $\mathcal{E}_2$, $\mathbf{merge}(\mathcal{E}_1, \mathcal{E}_2) = \mathcal{E}$, *where* $\mathcal{E}$ *is defined as follows:*

- *for all* $\mathfrak{A}_i \in \mathcal{E}_1$,
  - *if* $\mathbf{label}(\mathfrak{A}_i) \neq \mathbf{label}(\mathfrak{A}_j)$ *for all* $\mathfrak{A}_j \in \mathcal{E}_2$, *then* $\mathfrak{A}_i \in \mathcal{E}$,
  - *else if* $\mathbf{label}(\mathfrak{A}_i) = \mathbf{label}(\mathfrak{A}_j)$ *for some* $\mathfrak{A}_j \in \mathcal{E}_2$, *let* $\mathfrak{A}_i = \mathbf{inf}(x:\tau):\Sigma_i$, $\mathfrak{A}_j = \mathbf{inf}(x:\tau):\Sigma_j$ *and* $\Sigma = \mathbf{rectype}(\mathfrak{A}_i) = \mathbf{rectype}(\mathfrak{A}_j)$, *then* $\mathfrak{A}' \in \mathcal{E}$ *where:*

$$\begin{array}{rcl} \mathfrak{A}' & = & \mathbf{inf}(x:\tau):\Sigma' \textit{ and} \\ \Sigma' & = & (\Sigma_i - \Sigma_j) \cup (\Sigma_j - \Sigma_i) \cup \\ & & \{m:\tau' \mid m:\tau \in \Sigma, m:\tau_i \in \Sigma_i, m:\tau_j \in \Sigma_j, \tau' = \tau_i \sqcap_\tau \tau_j\}; \end{array}$$

- *for all* $\mathfrak{A}_k \in \mathcal{E}_2$ *such that* $\mathbf{label}(\mathfrak{A}_k) \neq \mathbf{label}(\mathfrak{A}_i)$, *for all* $\mathfrak{A}_i \in \mathcal{E}_1$, *then* $\mathfrak{A}_k \in \mathcal{E}$.

A key property of *merge* is its *monotonicity:* merging two type effects never decreases the inf associated to variables' types.

THEOREM 8 (SOUNDNESS OF THE ANNOTATION ALGORITHM) *If* $\mathbf{Ann}(\Gamma, P) = \langle \overline{P}, \mathbf{proc}, \mathcal{E} \rangle$ *then*

i) $\Gamma \vdash P : \mathbf{proc}$;

ii) for any free variable x of class or mixin type occurring in P, then there is one and only one type assertion $\mathfrak{A} \in \mathcal{E}$, *of the shape* $\mathbf{inf}(x:\tau):\Sigma'$, *such that:*

(a) (correctness of lower bounds) $\mathfrak{A}$ *is well defined (according to Definition 2);*

(b) (completeness of lower bounds) *for each* $m:\tau' \in \Sigma'$, m *is redefined by a mixin in some mixin application expression occurring in P and for each such redefinition* $m:\tau''$ *we have that* $\tau'' <: \tau'$.

COROLLARY 9 *For any well-typed closed process P,* $P \equiv \mathbf{receive}(x:\tau).P'$ ($P \equiv \mathbf{send}(A, \ell).P'$), *its compiled version* $\overline{P}$ *is of the form* $\overline{P} \equiv \mathbf{receive}(x^{\tau|\mathfrak{A}}).\overline{P'}$ ($\overline{P} \equiv \mathbf{send}(\overline{A}^{\tau}, \ell).\overline{P'}$), *where* $\mathfrak{A}$ *is correct and complete w.r.t. the occurrences of x in* P' (*A is of type* τ).

LEMMA 10 (SUBSTITUTION FOR EXPRESSIONS) *Let v and exp be an object-oriented value and an object-oriented expression, respectively. If* $\Gamma, x:\tau_1 \vdash exp : \tau$

and $\Gamma \vdash v : \tau_2$, *then* $\Gamma \vdash exp[v/x] : \tau'$, *provided that the following condition* (COND) *is satisfied:*

- *if* τ_1 *is a mixin or class type: if* $Ann(\Gamma, x:\tau_1, exp) = \langle exp, \tau, \mathcal{E} \rangle$ *and there is an assertion* $\mathfrak{A} \in \mathcal{E}$ *such that* $label(\mathfrak{A}) = x$, *then* $\tau_2 \models \mathfrak{A}$;
- *otherwise:* $\tau_2 <: \tau_1$.

THEOREM 11 (SUBSTITUTION FOR PROCESSES) *Let v, exp and P be an object-oriented value, an object-oriented expression and a process, respectively. If* $\Gamma, x:\tau_1 \vdash P : \mathsf{proc}$ *and* $\Gamma \vdash v : \tau_2$, *then* $\Gamma \vdash P[v/x] : \mathsf{proc}$, *provided that the following condition* (COND) *is satisfied:*

- *if* τ_1 *is a mixin or class type: if* $Ann(\Gamma, x:\tau_1, P) = \langle \overline{P}, \mathsf{proc}, \mathcal{E} \rangle$ *and there is an assertion* $\mathfrak{A} \in \mathcal{E}$ *such that* $label(\mathfrak{A}) = x$, *then* $\tau_2 \models \mathfrak{A}$;
- *otherwise:* $\tau_2 <: \tau_1$.

7. Operational Semantics

The operational semantics of MOMI groups two sets of rules. The first one describes how to evaluate SOOL object-oriented expressions and is denoted by $\rightarrowtail$. We omit it here since it is standard. The second set of rules, presented in Table 9, describes the evolution of a net. It is based on a standard structural congruence $\equiv$, defined as the least congruence relation closed under the following rules:

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

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

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

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 check dynamically that the received item is correct with respect to the formal argument. The other rules are straightforward.

Assuming that types are preserved under $\rightarrowtail$, a subject-reduction property is proved by using Theorem 11, that deals with the crucial case of rule *(comm)*. Then, the subject reduction property extends easily to a global type safety for nets, where a net N is *well typed* if and only if for any node $\ell :: \overline{P}$ in N, $\Gamma \vdash P : \mathsf{proc}$ for some Γ. Finally, the theorem below guarantees that merging (well-typed) code received from a remote site into local (well-typed) code does not harm local type safety.

THEOREM 12 (SUBJECT REDUCTION) *If N is well typed and* $N \rightarrowtail N'$, *then* N' *is well typed.*

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

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

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

Table 9. **Net and process operational semantics.**

The dynamic checking during communication is the only dynamic use of types: it consists essentially in checking some subtyping relations 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.

Let us go back to the example of the beginning of Section 6. The annotated versions of those processes are:

$\ell_1 :: \mathsf{send}(C^{\tau'},\ell_2)$, where $\tau' = \mathsf{class}\langle\{m:\mathsf{int}, n:\mathsf{posint}\}\rangle$, and

$\ell_2 :: \mathsf{receive}(x^{\tau \mid \mathsf{inf}(x:\tau):\{n:\mathsf{int}\}}).(\mathsf{new}\ M \diamond x) \Leftarrow m()$,

where $\tau = \mathsf{class}\langle\{m:\mathsf{int}, n:\mathsf{int}\}\rangle$.

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

8. Conclusions

We introduced a safe form of subtyping on classes and mixins, by offering a general and comprehensive solution both to "width subtyping versus addition" and "depth subtyping versus override" conflicts. Correctness is guaranteed by our renaming, to take care of name clashes, and by our constraints, to avoid accepting code that create override conflicts. The solution for the width-subtyping-related problem is the formal counter-part of classical implementation techniques to avoid name-clashes. The solution for the depth-subtyping-related problem is, at the best of our knowledge, the first proposal in the literature to solve such problem, and it is based on the simple observation that a method body cannot be overridden by a body whose type is bigger with respect to subtyping, i.e., is "less good".

Some future research directions look interesting: (*i*) to introduce *higher-order mixins* and *mixin composition* as presented in [17]; *(ii)* to replace structural subtyping with a form of nominal subtyping; *(iii)* to explore the possibility of applying a form of our "safe subtyping" to typed *traits* [21, 16].

In the literature, there are some proposals of combining objects with processes and/or mobile agents, such as, e.g., [13, 12]. Our approach is, however, more related to works as [22], where properties of distributed systems are enforced by a typing system equipped with subtyping. In our case the property we address is a flexible and type-safe coordination for exchanging code among processes.

Concerning the applicability of MOMI's approach, in [7] we presented O'KLAIM, a mixin-oriented version of KLAIM. A prototype implementation of O'KLAIM is presented in [4] and freely available at `http://music.dsi.unifi.it`. This is based on the Java package `momi` [3], that implements the run-time system (or the virtual

machine) for MOMI classes, mixins and objects. Code exchange in O'KLAIM exploits width subtyping only. An extended version of O'KLAIM (and relative implementation), including the annotation algorithm for dealing with depth subtyping, is work-in-progress.

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 *Proc. of ECOOP'00,* volume 1850 of *LNCS,* pages 145–178, 2000.

[3] L. Bettini. A Java package for class and mixin mobility in a distributed setting. In *Proc, of FIDJI'03,* volume 2952 of *LNCS,* pages 12–22. Springer-Verlag, 2003.

[4] L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations.* PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.

[5] L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In *Proc. of Coordination,* volume 2315 of *LNCS,* pages 56–71. Springer, 2002.

[6] L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Classes and Mixins. In *Proc. of FOOL 10,* 2003.

[7] L. Bettini, V. Bono, and B. Venneri. O'KLAIM: a coordination language with mobile mixins. In *Proc. of Coordination,* volume 2949 of *LNCS,* pages 20–37. Springer, 2004.

[8] 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-Verlag, 1995.

[9] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. of ECOOP'99,* volume 1628 of *LNCS,* pages 43–66. Springer-Verlag, 1999.

[10] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc.* of *OOPSLA '90,* pages 303–311. ACM, 1990.

[11] K. Bruce. *Foundations of Object-Oriented Languages –Types and Semantics.* The MIT Press, 2002.

[12] M. Bugliesi, S. Crafa, and G. Castagna. Typed Mobile Objects. In *Proc. of CONCUR '00,* volume 1877 of *LNCS,* pages 504–520. Springer-Verlag, 2000.

[13] L. Cardelli. A Language with Distributed Scope. *Computing Systems,* 8(1):27–59, 1995.

[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] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT'95,* volume 965 of *LNCS,* pages 42–61. Springer-Verlag, 1995.

[16] K. Fisher and J. Reppy. A typed calculus of traits. In *FOOL 11,* 2004.

[17] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. of POPL '98,* pages 171–183. ACM, 1998.

[18] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[19] B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[20] J. Riecke and C. Stone. Privacy via Subsumption. *Information and Computation,* 172:2–28, 2002.

[21] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *Proc. of ECOOP 2003,* volume 2743 of *LNCS,* pages 248–274. Springer, 2003.

[22] N. Yoshida and M. Hennessy. Subtyping and Locality in Distributed Higher Order Mobile Processes (extended abstract). In *Proc. of CONCUR'99,* volume 1664 of *LNCS,* pages 557–572. Springer-Verlag, 1999.