A Core Calculus of Higher-Order Mixins and Classes[⋆]

Lorenzo Bettini[1] Viviana Bono[2] Silvia Likavec[2]

[1] Dipartimento di Sistemi ed Informatica, Università di Firenze, Via Lombroso 6/17, 50134 Firenze, Italy, bettini@dsi.unifi.it

[2] Dipartimento di Informatica, Università di Torino, C.so Svizzera 185, 10149 Torino, Italy, {bono,likavec}@di.unito.it

Abstract. This work presents an object-oriented calculus based on *higher-order* mixin construction via *mixin composition*, where some software engineering requirements are modeled in a formal setting allowing to prove the absence of *message-not-understood* run-time errors. Mixin composition is shown to be a valuable language feature enabling a cleaner object-oriented design and development. In what we believe being quite a general framework, we give directions for designing a programming language equipped with higher-order mixins, although our study is not based on any already existing object-oriented language.

1 Introduction

Recently, mixins are undergoing a renaissance (see, for example, [7, 1, 8]), due to their flexible nature of "incomplete" classes prone to be completed according to the programmer's needs. Mixins [14, 19] are (sub)class definitions parameterized over a superclass and were introduced as an alternative to some forms of multiple inheritance [13, 22]. A mixin could be seen as a function that, given one class as an argument, produces another class, by adding or overriding certain sets of methods. The same mixin can be used to produce a variety of classes with the same functionality and behavior, since they all have the same sets of methods added and/or redefined. Also, the same mixin can sometimes be applied to the same class more than once, thus enabling incremental changes in the subclasses. The superclass definition is not needed at the time of writing the mixin definition. This minimizes the dependences between superclass and its subclasses, as well as between class implementors and end-users, thus improving modularity. The uniform extension and modification of classes is instead absent from the classical class-based languages. In this work we extend the core calculus of classes and mixins of [10] with *higher-order* mixins. A mixin can: (*i*) be applied to a class to create a fully-fledged subclass; or (and this is the novelty with respect to [10]) (*ii*) be *composed* with another mixin to obtain yet another mixin with more functionalities. In Section 2.1 we present some uses of mixin inheritance and, in particular, we show that mixin composition enables a cleaner modular object-oriented design.

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

This paper presents a framework for the construction of composite mixins, and therefore of sophisticated class hierarchies, while keeping the good features of the original core calculus of [10]. In particular, we retain *structural subtyping*. As in most popular object-oriented languages, objects in our calculus can only be created by instantiating a class. We use structural subtyping to remove the dependency of object users on class implementation. Each object has an *object type*, which lists the names and types of methods and fields but does *not* include information about the class from which the object was instantiated. Therefore, objects created from unrelated classes can be substituted for each other if their types satisfy the subtyping relation. Structural subtyping was a deliberate design decision already in [10, 11, 24], motivated by the desire to minimize code dependencies between object users and class implementors. A different approach would be to follow Java or C++, in which an object's type is related to the class from which it was instantiated, and subtyping relations apply only to objects instantiated from the same class hierarchy (*nominal subtyping*). Subtyping is defined on object types only, not on class and mixin types, to avoid the well-known inheritance-subtyping conflicts (for an account on the subject, see for instance [15]). As a consequence of the absence of subtyping on classes, a higher-order mixin is more than a function that consumes and produces classes, since such a function cannot accept a class with extra methods as an argument. Moreover, the type system would have to express that the result of the "mixin-function" has at least the methods of the argument, and such general extensions to the type system look unnecessarily complex for the model's more specific purpose.

Our design decisions are strongly based on the choices that were made in [10]. Class hierarchies in a well-designed object-oriented program must not be fragile: if a superclass implementation changes but the specification remains intact, the implementors of the subclasses should not have to rewrite subclass implementations. This is only possible if object creation is modular. In particular, a subclass implementation should not be responsible for initializing inherited fields when a new object is created, since some of the inherited fields may be private and thus invisible to the subclass. Also, the definitions of inherited fields may change when the class hierarchy changes, making the subclass implementation invalid. Unlike many theoretical calculi for object-oriented languages, our calculus directly supports modular object construction. The mixin implementor only writes the local constructor for his own mixin. Mixin applications and compositions are reduced to generator functions that call all constructors in the inheritance chain in the correct order, producing a fully initialized object (see Section 3). Unlike some approaches to encapsulation in object calculi such as existential types, the levels of encapsulation describe *visibility*, and not merely *accessibility*. For example, even the names of private items are invisible outside the class in which they are defined. This seems to be a better approach since *no* information about data representation is revealed, not even the number and names of fields. One of the benefits of using visibility-based encapsulation is that no conflicts arise if both the superclass and the subclass declare a private field with the same name. Among other advantages, this allows the same mixin to be applied twice (see the example in Section 2.1). To ensure that mixin inheritance can be statically type checked, the calculus employs constrained parameterization. From each mixin definition the type system infers a constraint specifying to which classes the mixin can be applied so that the resulting subclass is type-safe. The constraint includes

$$\begin{array}{ll}
\begin{aligned}
e ::= {} & \mathit{const} \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathit{fix} \\
& \mid \mathsf{ref} \mid\ !\ \mid\ := \mid \{x_i = e_i\}^{i\in I} \mid e.x \\
& \mid \mathsf{H}\ h.e \mid \mathsf{classval}\langle v_g, \mathcal{M}\rangle \mid \mathsf{new}\ e \\
& \mid \mathsf{mixin} \\
& \quad \mathsf{method}\ m_j = v_{m_j};\ ^{(j\in New)} \\
& \quad \mathsf{redefine}\ m_k = v_{m_k};\ ^{(k\in Redef)} \\
& \quad \mathsf{expect}\ m_i;\ ^{(i\in Expect)} \\
& \quad \mathsf{constructor}\ v_c; \\
& \quad \mathsf{end} \\
& \mid \mathsf{mixinval}\langle v_m, New, Redef, Expect\rangle \\
& \mid e_1 \diamond e_2 \mid e_1 \bullet e_2
\end{aligned}
&
\begin{aligned}
v ::= {} & \mathit{const} \mid x \mid \lambda x.e \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \\
& \mid\ := \ \mid\ := v \mid \{x_i = v_i\}^{i\in I} \\
& \mid \mathsf{classval}\langle v_g, \mathcal{M}\rangle \\
& \mid \mathsf{mixinval}\langle v_m, New, Redef, Expect\rangle
\end{aligned}
\end{array}$$

Fig. 1. Syntax of the core calculus: expressions and values.

both *positive* (which methods the class must contain) and *negative* (which methods the class must not contain) information. New and redefined methods are distinguished in the mixin implementation: from the implementor's viewpoint, a new method may have arbitrary behavior, while the behavior of a redefined method must be "compatible" with that of the old method it replaces. Having this distinction in the syntax of our calculus helps mixin implementors avoid unintentional redefinitions of superclass methods and facilitates generation of the constraint for mixin's superclasses and for mixins that participate in mixin composition (see Section 4). A marginal difference with respect to the original mixin calculus [10] is that we do not treat *protected* methods, being an orthogonal issue to higher-order mixins. Nevertheless, protected methods could be easily accounted for via (structural) subtyping as in the original calculus.

2 Syntax of the Calculus

The starting point for our calculus is the core calculus of classes and mixins of Bono et al. [10] that, in turn, is based on *Reference ML* of Wright and Felleisen [25]. To this imperative calculus of records and functions, we add constructs for manipulating classes and mixins. The class and mixin related expressions are: classval, mixin, mixinval, $\diamond$ (mixin application), $\bullet$ (mixin composition) and new. The novelties with respect to [10] are mixinval and $\bullet$ (mixin composition) to deal with *higher-order* mixins.

Expressions and values are given in Figure 1. Most of them are standard, the only constructs that might need some explanation are the following:

– ref, !, $:=$ are operators[3] for defining a reference to a value, for de-referencing a reference, and for assigning a new value to a reference, respectively.
– $\{x_i = e_i\}^{i\in I}$ is a record and $e.x$ is the record selection operation (note that this corresponds to method selection in our calculus).
– h is a set of pairs $h ::= \{\langle x, v\rangle^*\}$ where x is a variable and v is a value (first components of the pairs are all distinct). We have a concept of a *heap*, represented by h in the expression $\mathsf{H}h.e$, used for evaluating imperative side effects. In the expression $\mathsf{H}\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$, H binds variables $x_1, \ldots x_n$ in $v_1, \ldots, v_n$ and in e.

[3] Introducing ref, !, $:=$ as operators rather than standard forms such as ref$e$, !e, $:=e_1e_2$, simplifies the definition of evaluation contexts and proofs of properties. As noted in [25], this is just a syntactic convenience, as is the curried version of $:=$.

- new e uses generator v_g of the class value to which e evaluates to create a function that returns a new object, as described in Section 3.
- classval$\langle v_g, \mathcal{M} \rangle$ is a *class value*, and it is the result of mixin application. It is a pair, containing the function v_g, that is the generator for the class used to generate its instance objects, and the set $\mathcal{M}$ of the indices of all the methods defined in the class. In our calculus method names are of the shape m_i, where i ranges over an index set, and are univocally identified by their index, i.e., $m_i = m_j$ if and only if $i = j$.
- mixin

  method $m_j = v_{m_j}$; $^{(j \in New)}$

  redefine $m_k = v_{m_k}$; $^{(k \in Redef)}$

  expect m_i; $^{(i \in Expect)}$

  constructor v_c;

  end

  is a *mixin* expression, and it states the methods that are new, redefined, and expected in the mixin (names of which have to be all distinct). More precisely, $m_j = v_{m_j}$ are definitions of the new methods, $m_k = v_{m_k}$ are method redefinitions that will replace the methods with the same name in the superclass, and m_i are method (names) that the superclass is expected to implement. Each method body v_{m_j} (respectively, v_{m_k}) is a function of the private *field* and of *self*, which will be bound to the newly created object at instantiation time. In method redefinitions, v_{m_k} is also a function of *next*, which will be bound to the corresponding old method from the superclass. The v_c value in the constructor clause is a function that returns a record of two components: the fieldinit value is used to initialize the private field; the superinit value is passed as an argument to the superclass constructor. When evaluating a mixin, v_c is used to build the generator as described in Section 3.
- mixinval$\langle v_m, New, Redef, Expect \rangle$ is a *mixin value*, and it is the result of a mixin evaluation. It is a tuple, containing one function and three sets of indices. The function v_m is the (partial) generator for the corresponding mixin. The sets *New*, *Redef*, and *Expect* contain the names of all methods defined in the mixin (new, redefined, and expected).
- $e_1 \diamond e_2$ denotes the application of mixin value e_1 to class value e_2. Given the (super)class value e_2 as an "argument" to e_1, it produces a new (sub)class value.
- $e_1 \bullet e_2$ is a composition of two mixin values e_1 and e_2. It produces a new mixin value taking components from both e_1 and e_2. The resulting mixin can be applied to class values to produce new classes, as well as composed with other mixin values to produce new composite mixins.

As in [10], we define the root of the class hierarchy, class *Object*, as a predefined class value: $Object \stackrel{\triangle}{=}$ classval$\langle\, \lambda_.\lambda_.\{\},\ [\ ]\ \rangle$. The root class is necessary so that all other classes can be treated uniformly and it is the only class value that is not obtained as a result of mixin application. The calculus can then be simplified by assuming that any user-defined class that does not need a superclass is obtained by applying a mixin containing all of the class method definitions to *Object*. For the sake of clarity, in the following examples we will avoid the explicit mixin application to *Object*.

2.1 An example of mixin inheritance

In this section, we present a simple example that shows how mixins can be implemented and used in our calculus and explain some of the uses of mixin application and mixin composition. For readability, the example uses functions with multiple arguments even though they are not formalized explicitly in the calculus.

In the following, we give the definitions of `Encrypted` mixin and `Compress` mixin that implement encryption and compression functionality on top of any stream class, respectively. Note that the class to which the mixin is applied may have more methods than expected by the mixin. For example, `Encrypted` can be applied to `Socket` $\diamond$ *Object*, even though `Socket` $\diamond$ *Object* has other methods besides *read* and *write*. The mixin `Random` allows random access to any stream class, thus we can build a random access file class with the mixin application `Random` $\diamond$ `FileStream`.

```
let FileStream = mixin      let Socket = mixin            let Random = mixin
  method write = ...          method write = ...            method lseek = ...
  method read = ...           method read = ...             expect write;
 end in                       method IPaddress = ...        expect read;
                             end in                        end in

   let Encrypted =
    mixin
     redefine write = λ key. λ self. λ next. λ data. next (encrypt(data, key));
     redefine read = λ key. λ self. λ next. λ _ . decrypt(next (), key);
     constructor λ (key, arg). {fieldinit=key, superinit=arg};
    end in
 let Compress =
  mixin
   redefine write = λ level. λ self. λ next. λ data. next (compress(data, level));
   redefine read = λ level. λ self. λ next. λ _ . uncompress(next (), level);
   constructor λ (level, arg). {fieldinit=level, superinit=arg};
  end in ...
```

From the definition of `Encrypted`, the type system infers the types of the methods that the mixin wants to redefine. These are the constraints that must be satisfied by any class to which `Encrypted` is applied. The class must contain *write* and *read* methods whose types must be supertypes of those given to *write* and *read*, respectively, in the definition of `Encrypted`. In `Random` such methods are declared as *expected* and they are used within the method *lseek*. Once again the type system infers their types according to how they are used in *lseek*.

To create an encrypted stream class, one must apply the `Encrypted` mixin to an existing stream class. For example, `Encrypted` $\diamond$ `FileStream` is an encrypted file class. The power of mixins can be seen when we apply `Encrypted` to a family of different streams. For example, we can construct `Encrypted` $\diamond$ `Socket`, which is a class that encrypts data communicated over a network. In addition to single inheritance, we can express many uses of multiple inheritance by applying more than one mixin to a class. For example, `PGPSign` $\diamond$ `UUEncode` $\diamond$ `Encrypted` $\diamond$ `Compress` $\diamond$ `FileStream` produces a class of files that are compressed, then encrypted, then uuencoded, then signed.

In addition, mixins can be used for forms of inheritance that are not possible in most single and multiple inheritance-based systems. In the above example, the result of applying `Encrypted` to a stream satisfies the constraint required by `Encrypted` itself, therefore, we can apply `Encrypted` more than once: `Encrypted` $\diamond$ `Encrypted` $\diamond$ `FileStream` is a class of files that are encrypted twice. In our calculus, class private fields do not conflict even if they have the same name, so each application of `Encrypted` can have its own encryption key.

Mixin composition further enhances the (re)usability of classes and mixins and enables better modular programming design, by exploiting software composition at a higher level. For example, the programmer is able to build a customized library of reusable mixins starting from existing mixins: one can create the new mixin `2Encrypt` = `Encrypted` $\bullet$ `Encrypted`, instead of always applying the mixin `Encrypted` twice to every stream class in her program. This also enables consistency: if in the future the definition of the mixin `2Encrypt` must be extended, e.g., by also exploiting UU encoding, then by changing only the definition of `2Encrypt`, with an additional mixin composition, it is guaranteed that all the functions that used `2Encrypt` will use the new version. Moreover, construction of mixins can be delegated to different parts of the program (thus exploiting modular programming), and the resulting mixins can then be assembled in order to build a class. For instance, the following code delegates the construction of mixins for encryption and compression to two functions, and then assembles the returned mixins for later use:

$$\begin{array}{l}\mathsf{let}\ m_1 = \mathrm{build\_compression}()\ \ \mathsf{in}\ \mathsf{let}\ m_2 = \mathrm{build\_encryption}()\ \ \mathsf{in}\\ \quad \mathsf{let}\ m = m_1 \bullet m_2\ \ \mathsf{in}\ (\mathrm{new}(m \diamond \mathrm{FileStream})).\mathrm{write}(\texttt{"foo"})\end{array}$$

The function `build_compression` returns a specific mixin according to user's requests: it can return a simple `Compress` mixin, or a more elaborate `UUEncode` $\bullet$ `Compress` mixin. Similarly, `build_encryption`, instead of simply returning a mixin `Encrypted`, returns the composition `PGPSign` $\bullet$ `Encrypted`. All these enhanced modular composition functionalities, supported by mixin composition, would not be directly provided by simple mixin application.

Finally, let us observe that streams are implemented usually via the design pattern *decorator* [21] (for instance, in Java), and this requires additional manual programming. Instead, with mixins (and in particular with mixin composition), streams can be programmed directly exploiting language features. This is just one of the examples of the additional expressiveness provided by mixin composition.

3 Operational Semantics

The operational semantics of the original calculus [10] is very close to an implementation, and we follow the same approach. Our operational semantics is a set of rewriting rules including the standard rules for a lambda calculus with stores (in our case the Reference ML [25]), and some rules that evaluate the object-oriented related forms to records and functions, following the "objects-as-records" technique and Cook's "class-as-generator-of-object" principle. This operational semantics can be seen also as something extremely close to a denotational description for objects, classes, and mixins, and this "identification" of implementation and semantical denotation is, according to us, a good by-product of our approach.

$$
\begin{array}{rlcrll}
\mathit{const}\ v \longrightarrow & \delta(\mathit{const},v) & (\delta) & \mathsf{ref}\,v \longrightarrow & \mathsf{H}\langle x,v\rangle.x & (\mathrm{ref}) \\
\text{if} & \delta(\mathit{const},v) & \text{is defined} & \mathsf{H}\langle x,v\rangle h.R[!x] \longrightarrow & \mathsf{H}\langle x,v\rangle h.R[v] & (\mathrm{deref}) \\
(\lambda x.e)\ v \longrightarrow & [v/x]\ e & (\beta_v) & \mathsf{H}\langle x,v\rangle h.R[:=xv'] \longrightarrow & \mathsf{H}\langle x,v'\rangle h.R[v'] & (\mathrm{assign}) \\
\mathit{fix}\ (\lambda x.e) \longrightarrow & [\mathit{fix}(\lambda x.e)/x]e & (\mathrm{fix}) & R[\mathsf{H}\ h.e] \longrightarrow & \mathsf{H}\ h.R[e],\ \ R \neq [\,] & (\mathrm{lift}) \\
\{\ldots,x=v,\ldots\}.x \longrightarrow & v & (\mathrm{select}) & \mathsf{H}\ h.\mathsf{H}\ h'.e \longrightarrow & \mathsf{H}\ h\ h'.e & (\mathrm{merge})
\end{array}
$$

$$
\begin{pmatrix}
\mathsf{mixin} \\
\ \mathsf{method}\ m_j = v_{m_j}; \\
\ \mathsf{redefine}\ m_k = v_{m_k}; \\
\ \mathsf{expect}\ m_i; \\
\ \mathsf{constructor}\ c; \\
\mathsf{end}
\end{pmatrix}^{\substack{j \in \mathit{New} \\ k \in \mathit{Redef} \\ i \in \mathit{Expect}}}
\longrightarrow \mathsf{mixinval}\langle \mathit{Gen}_m, \mathit{New}, \mathit{Redef}, \mathit{Expect}\rangle \qquad (\mathrm{mixval})
$$

$$
\begin{array}{l}
\mathit{Gen}_m \stackrel{\triangle}{=} \lambda x. \\
\quad \mathsf{let}\ t = c(x) \quad \mathsf{in} \\
\qquad \left\{ \begin{array}{l}
\mathsf{gen} = \lambda \mathit{self}. \\
\left\{ \begin{array}{ll}
m_j = \lambda y.v_{m_j}\ t.\mathsf{fieldinit}\ \mathit{self}\ y & {}^{j \in \mathit{New}} \\
m_k = \lambda y.v_{m_k}\ t.\mathsf{fieldinit}\ \mathit{self}\ y & {}^{k \in \mathit{Redef}}
\end{array} \right\}, \\
\mathsf{superinit} = t.\mathsf{superinit}
\end{array} \right\}
\end{array}
$$

$$
\mathsf{mixinval}\langle \mathit{Gen}_m, \mathit{New}, \mathit{Redef}, \mathit{Expect}\rangle \diamond \mathsf{classval}\langle g, \mathcal{M}\rangle \longrightarrow \mathsf{classval}\langle \mathit{Gen}, \mathit{New} \cup \mathcal{M}\rangle \quad (\mathrm{mixapp})
$$

$$
\begin{array}{l}
\mathit{Gen} \stackrel{\triangle}{=} \lambda x.\lambda \mathit{self}. \\
\quad \mathsf{let}\ \mathit{mixinrec} = \mathit{Gen}_m(x) \quad \mathsf{in} \\
\quad \mathsf{let}\ \mathit{mixingen} = \mathit{mixinrec}.\mathsf{gen} \quad \mathsf{in} \\
\quad \mathsf{let}\ \mathit{supergen} = g(\mathit{mixinrec}.\mathsf{superinit}) \quad \mathsf{in} \\
\qquad \left\{ \begin{array}{ll}
m_j = \lambda y.(\mathit{mixingen}\ \mathit{self}).m_j\ y & {}^{j \in \mathit{New}} \\
m_k = \lambda y.(\mathit{mixingen}\ \mathit{self}).m_k\ (\mathit{supergen}\ \mathit{self}).m_k\ y & {}^{k \in \mathit{Redef}} \\
m_i = \lambda y.(\mathit{supergen}\ \mathit{self}).m_i\ y & {}^{i \in \mathcal{M} - \mathit{Redef}}
\end{array} \right\}
\end{array}
$$

Fig. 2. Reduction rules

$$
\begin{array}{l}
R ::= [\ ] \mid R\ e \mid v\ R \mid R.x \mid \mathsf{new}\ R \mid R \diamond e \mid v \diamond R \mid R \bullet e \mid v \bullet R \\
\quad \mid \{m_1 = v_1, \ldots, m_{i-1} = v_{i-1}, m_i = R, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \leq i \leq n}
\end{array}
$$

Fig. 3. Reduction contexts

The operational semantics extends the one of the core calculus of classes and mixins, [10], and therefore exploits the *Reference ML* of Wright and Felleisen treatment of side-effects [25]. We give the reduction rules in Figures 2 and 4. To abstract from a precise set of constants, we only assume the existence of a partial function $\delta : \mathit{Const} \times \mathit{ClosedVal} \rightharpoonup \mathit{ClosedVal}$ that interprets the application of functional constants to closed values and yields closed values. In Figure 2, R are the *reduction contexts* [23, 17, 18]. Reduction contexts are necessary to provide a minimal relative linear order among the creation, dereferencing and updating of heap locations, since side effects need to be evaluated in a deterministic order. Their definition can be found in Figure 3. We assume the reader is familiar with the treatment of imperative side-effects via reduction contexts and we refer to [25, 10] for a description of the related rules.

(*new*) rule is responsible for instantiating new objects from class definitions. The resulting function can be thought of as the composition of two functions: $\mathit{fix} \circ g$. First, the generator g is applied to an argument v, thus creating a function from *self* to a record of methods. Afterwards, the fixed-point operator *fix* is applied to bind *self* in method bodies and create a recursive record (following [16]). The resulting record is a fully formed object that could be returned to the user.

Rule (*mixval*) turns a mixin expression into a mixin *value*. A mixin value consists of a mixin generator Gen_m and of the sets of mixin method names (new, redefined, and expected; we recall that names are identified with their indices, as said in Section 2). Gen_m is a sort of a compiled (equivalent) version of the mixin expression. Given the parameter for the mixin constructor c, Gen_m returns a record containing a (partial) object generator gen, and the argument superinit for the (future) superclass constructor. We recall that c is a function of one argument which returns a record of two components: one is the initialization expression for the method field (fieldinit), the other is the superclass generator's argument (superinit). The object generator gen binds the private field of the methods defined (*New*) and redefined (*Redef*) by the mixin to fieldinit (recall that method bodies take parameters for *field*, for *self*, and, if the method is a redefinition, also for *next*, which will be bound to the corresponding superclass method). The returned object generator is partial because it comes from a mixin, i.e., the expected methods and the *next* for each redefined method will be provided by a superclass or by other mixins (in fact, note that *next* is not yet bound in m_k's bodies). Notice that all the other mixin operations, i.e., mixin application and mixin composition, are performed on mixin values. In the original calculus of [10], mixin values are created and "blended" directly at mixin-application time with a (super)class value to obtain a (sub)class value. Here mixin values are made explicit to deal smoothly with mixin composition. For all the methods, the method bodies are wrapped inside $\lambda y. \cdots y$ to delay evaluation in our call-by-value calculus.

Rule (*mixapp*) evaluates the application of a mixin value to a class value, performing mixin-based inheritance. A mixin value $\mathsf{mixinval}\langle Gen_m, New, Redef, Expect\rangle$ is applied to a class value $\mathsf{classval}\langle g, \mathcal{M}\rangle$ which plays the rôle of the superclass, where g is the object generator of the superclass and $\mathcal{M}$ is the set of all method names defined in the superclass. The resulting class value is $\mathsf{classval}\langle Gen, New \cup \mathcal{M}\rangle$, where *Gen* is the generator function for the subclass, and $New \cup \mathcal{M}$ lists all its method names. Using a class generator delays full inheritance resolution until object instantiation time when *self* becomes available. The generator *Gen* takes a single argument x, which is used by the mixin generator, and returns a function from *self* to a record of methods. When the fixed-point operator is applied to the function returned by the generator, it produces a recursive record of methods representing a new object (see rule (*new*)). *Gen* first calls $Gen_m(x)$ to compute the mixin object generator *mixingen*, a function from *self* to a record of mixin methods, and the parameter *mixingen*.superinit to be passed to the superclass generator g, that, in turn, returns a function *supergen* from *self* to a record of superclass methods. *Gen* results to be a function of *self* that returns a record containing *all* the methods — from both the mixin and the superclass. All methods of the superclass that are not redefined by the mixin, m_i where $i \in \mathcal{M} - Redef$, are *inherited* by the subclass: they are taken intact from the superclass's "object" (*supergen self*). These methods m_i include all the methods that are expected by the mixin (this is ensured by the type system, see Section 4). Methods m_j defined by the mixin are taken intact from the mixin's "object" (*mixingen self*). As for *redefined* methods m_k, *next* is bound to

$$
\begin{array}{l}
\mathsf{mixinval}\langle g_1, New_1, Redef_1, Expect_1\rangle \bullet \mathsf{mixinval}\langle g_2, New_2, Redef_2, Expect_2\rangle \rightarrow \\
\qquad \mathsf{mixinval}\langle Gen, New_1 \cup New_2, (Redef_1 \cup Redef_2) - New_2, \\
\qquad\quad (Expect_1 - (New_2 \cup Redef_2)) \cup (Expect_2 - Redef_1)\rangle
\end{array}
$$

$$
\begin{array}{l}
Gen \overset{\triangle}{=} \lambda x. \\
\quad \mathsf{let}\ leftrec = g_1(x)\quad \mathsf{in} \\
\quad \mathsf{let}\ rightrec = g_2(leftrec.\mathsf{superinit})\quad \mathsf{in} \\
\quad \mathsf{let}\ leftgen = leftrec.\mathsf{gen}\quad \mathsf{in} \\
\quad \mathsf{let}\ rightgen = rightrec.\mathsf{gen}\quad \mathsf{in} \\
\quad \left\{ \begin{array}{l}
\mathsf{gen} = \lambda self. \\
\left\{ \begin{array}{ll}
m_{j_1} = \lambda y.(leftgen\ self).m_{j_1}\ y & {}^{j_1 \in New_1} \\
m_{j_2} = \lambda y.(rightgen\ self).m_{j_2}\ y & {}^{j_2 \in New_2 - Redef_1} \\
m_{j_3} = \lambda y.(leftgen\ self).m_{j_3}\ (rightgen\ self).m_{j_3}\ y & {}^{j_3 \in Redef_1 \cap New_2} \\
m_{k_1} = \lambda y.(leftgen\ self).m_{k_1}\ y & {}^{k_1 \in Redef_1 - (New_2 \cup Redef_2)} \\
m_{k_2} = \lambda next.(leftgen\ self).m_{k_2}\ ((rightgen\ self).m_{k_2}\ next) & {}^{k_2 \in Redef_1 \cap Redef_2} \\
m_{k_3} = \lambda y.(rightgen\ self).m_{k_3}\ y & {}^{k_3 \in Redef_2 - Redef_1}
\end{array} \right\}, \\
\mathsf{superinit} = rightrec.\mathsf{superinit}
\end{array} \right\}
\end{array}
$$

Fig. 4. Reduction rule (mixcomp) for mixin composition

$(supergen\ self).m_k$ in Gen. Notice that, at this stage, all methods have already received a binding for the private field. The variable *self* is passed all along in all method forms, in such a way that the host object will be bound appropriately at object creation time.

Rule (*mixcomp*) (Fig. 4) composes two mixins to produce a new mixin. The two mixins may partially complete each others' definitions, providing (some of) the missing components. Let us denote the mixin composition by $e_1 \bullet e_2$ and the resulting mixin by e. When composing two mixins, it is necessary to determine which sets of new/redefined/expected methods the new mixin e will have. Our design decision is as follows: the mixin e_2 acts as a "superclass" for e_1 (mirroring mixin application order), and, in particular, some of e_1 methods may override some of e_2 methods. Therefore, all the new methods of the mixin e_1 (New_1) are inserted in the resulting mixin e, while only the new methods of e_2 that are not redefined by e_1 ($j_2 \in New_2 - Redef_1$) become part of the new mixin. Notice that the type rule for mixin composition (*mixin comp*) (Figure 6) must check that no name clashes between new methods of e_1 and any method of e_2 take place. This decision is in line with a good object-oriented design principle of not confusing method redefinitions and name clashes. Therefore, an error is signaled at compile time and not at runtime. As far as redefined methods are concerned, the situation is more complex: the methods specified as redefining in e_1 can override some new methods of e_2, some redefining methods of e_2, and (even if only virtually) some of the expected methods of e_2.

– If a method m_{j_3} in e_1 redefines a method defined in e_2 ($j_3 \in Redef_1 \cap New_2$), then the overriding is completed and m_{j_3} becomes a new method in the resulting mixin e, after binding its *next* to e_2's implementation of m_{j_3};
– If e_1 redefines a method m_{k_2} that, in turn, is redefined by e_2 ($k_2 \in Redef_1 \cap Redef_2$), then this method is still a redefined method in e. Since e_1 "overrides" e_2, therefore m_{k_2}'s implementation of e_1 redefines that of e_2, the *next* in the implementation of e_1 is bound to the implementation of e_2, and the *next* in the implementation of e_2 is not bound, since it will be bound during future mixin composition or mixin

application. This means that the redefinition of a method m_{k_2} by means of e_2 is delayed (while e_1 has already performed its "internal" redefinition of m_{k_2} over e_2);
- If e_1 redefines a method that is expected in e_2, then this method will become a redefined method in e, so it will not appear among the expected methods of e, but it will be a method that e is willing to redefine.

Apart from the above examined methods, method redefinitions that are still present as method redefinitions in the resulting mixin e are: (*i*) the redefining ones from e_2 that are not redefined by e_1 ($k_3 \in Redef_2 - Redef_1$); (*ii*) the ones from e_1 that are not defined in e_2 and hence not "overriding" anything yet ($k_1 \in Redef_1 - (New_2 \cup Redef_2)$).

Finally, new and redefined methods from e_2 can provide some of the definitions that the mixin e_1 expects; in that case, such methods expected by e_1 do not appear anymore in the expected method set of e.

The generator of the new mixin is a combination of the generators of e_1 and e_2. Since e_1 is considered to be the "subclass", the parameter x is passed to g_1, and g_2 receives as a parameter the superinit returned by $g_1(x)$; the superinit field of the record returned by the generator of the new mixin is set to $g_2(g_1(x).\mathsf{superinit}).\mathsf{superinit}$. This strategy for building the new mixin generator corresponds to serializing the call of the two constructors similarly to what happens in standard object-oriented languages. Notice that this is consistent with the type $\mathsf{mixin}\langle\gamma_{b_2}, \gamma_{d_1}, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle$ assigned to the new mixin by the type rule (*mixin comp*) (Figure 6).

4 Type System

In addition to functional, record, and reference types of *Reference ML* type system, our type system has class-types and mixin-types.

The types in our system are the following:

$$\tau ::= \iota \mid \tau_1 \to \tau_2 \mid \tau\ \mathsf{ref} \mid \{m_i : \tau_{m_i}\}^{i \in I} \mid \mathsf{class}\langle\tau, \Sigma_b\rangle \mid \mathsf{mixin}\langle\tau_1, \tau_2, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle$$

where ι is a constant type, $\to$ is the functional type operator, $\tau\ \mathsf{ref}$ is the type of locations containing a value of type τ. The other type forms are described below.

Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i}\}^{i \in I}$. The set of indexes I (where $I \subseteq \mathbb{N}$) is often omitted when it is not relevant. A record type can be viewed as a set of pairs *label*:*type* where labels are pairwise disjoint (Σ_1 and Σ_2 are considered *equal*, denoted by $\Sigma_1 = \Sigma_2$, if they differ only in the order of their elements). 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 Σ (with type τ_{m_i}). $Subj(\Sigma)$ denotes the set of all subjects occurring in Σ;
- $\Sigma_1 \cup \Sigma_2$ is the standard set union (used only on Σ_1 and Σ_2 such that $Subj(\Sigma_1) \cap Subj(\Sigma_2) = \emptyset$, in order to guarantee that $\Sigma_1 \cup \Sigma_2$ is a record type);
- $\Sigma_1 - \Sigma_2$ is the standard set difference;
- $\Sigma_1 / \Sigma_2 = \{m_i : \tau_{m_i} \mid m_i : \tau_{m_i} \in \Sigma_1 \ \wedge\ m_i \text{ occurs in } \Sigma_2\}$.

The definitions of typing environments Γ and of typing judgments are standard. Our type system supports *structural subtyping* ($<:$ relation) along with a subsumption rule (*sub*). The subtyping rules are shown in Appendix A. Since subtyping on references is unsound and we wish to keep subtyping and inheritance completely separate, we have only the basic subtyping rules for function and record types. Subtyping only exists at the object level, and is not supported for class or mixin types (as explained in the introduction).

In the class type $\mathsf{class}\langle\gamma,\Sigma_b\rangle$, γ is the type of the generator's argument and $\Sigma_b = \{m_i : \tau_{m_i}\}$ is a record type representing *self*.

In the mixin type $\mathsf{mixin}\langle\gamma_b,\gamma_d,\Sigma_{new},\Sigma_{red},\Sigma_{exp},\Sigma_{old}\rangle$

- γ_b is the expected argument type of the superclass generator,
- γ_d is the exact argument type of the mixin generator,
- $\Sigma_{new} = \{m_j : \tau^{\downarrow}_{m_j}\}$ are the exact types of the new methods introduced by the mixin,
- $\Sigma_{red} = \{m_k : \tau^{\downarrow}_{m_k}\}$ are the exact types of the methods redefined by the mixin,
- $\Sigma_{exp} = \{m_i : \tau^{\uparrow}_{m_i}\}$ are the types of the methods that are neither defined nor redefined by the mixin, but expected to be supported by a superclass which the mixin will be applied to, or by another mixin which the mixin will be composed with,
- $\Sigma_{old} = \{m_k : \tau^{\uparrow}_{m_k}\}$ are the types assumed for the old bodies of the methods redefined by the mixin.

We report in Figure 5 the typing rules regarding classes and mixins (the rest of the typing rules are given in Appendix A). Some of them are syntactic variations of those presented in [10] and we refer the reader to that paper for comments about such rules. We only comment upon the rules related to mixin forms. The rules (*mixin*) and (*mixin val*) assign the same type to their respective expressions, although deduced in a different way. In the rule (*mixin*) the side condition $Subj(\Sigma_{new}) \cap Subj(\Sigma_{red}) \cap Subj(\Sigma_{exp}) = \emptyset$ ensures that the names of new, redefined, and expected methods are all distinct. In the rule (*mixin app*), Σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied, and Σ_b/Σ_{red} are the superclass methods redefined by the mixin (the superclass may have more methods than those required by the mixin constraints). The premises of the rule (*mixin app*) are the following:

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

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

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

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

Now we concentrate on the main topic of the paper, the rule for mixin composition (*mixin comp*) given in Figure 6. Since e_2 acts as the "superclass" of e_1, e_1 will pass the

$$\frac{\Gamma \vdash g : \gamma \rightarrow \{m_i : \tau_{m_i}\}^{i \in \mathcal{M}} \rightarrow \{m_i : \tau_{m_i}\}^{i \in \mathcal{M}}}{\Gamma \vdash \mathsf{classval}\langle g, \mathcal{M}\rangle : \mathsf{class}\langle \gamma, \{m_i : \tau_{m_i}\}^{i \in \mathcal{M}}\rangle}\ (\textit{class val}) \qquad \frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \{m_i : \tau_{m_i}\}\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \rightarrow \{m_i : \tau_{m_i}\}}\ (\textit{instantiate})$$

$$\frac{\begin{array}{rl} (\mathsf{New})\ \text{For } j \in \textit{New}: & \Gamma \vdash v_{m_j} : \eta \rightarrow \Sigma \rightarrow \tau^{\downarrow}_{m_j} \\ (\mathsf{Redef})\ \text{For } k \in \textit{Redef}: & \Gamma \vdash v_{m_k} : \eta \rightarrow \Sigma \rightarrow \tau^{\uparrow}_{m_k} \rightarrow \tau^{\downarrow}_{m_k} \\ (\mathsf{Constr}) & \Gamma \vdash c : \gamma_d \rightarrow \{\text{fieldinit} : \eta, \text{superinit} : \gamma_b\} \\ & \textit{Subj}(\Sigma_{new}) \cap \textit{Subj}(\Sigma_{red}) \cap \textit{Subj}(\Sigma_{exp}) = \emptyset \end{array}}{\Gamma \vdash \begin{pmatrix} \mathsf{mixin} \\ \ \mathsf{method}\ m_j = v_{m_j}; \\ \ \mathsf{redefine}\ m_k = v_{m_k}; \\ \ \mathsf{expect}\ m_i; \\ \ \mathsf{constructor}\ c; \\ \mathsf{end} \end{pmatrix}^{\substack{j \in New \\ k \in Redef \\ i \in Expect}} : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle}\ (\textit{mixin})$$

$$\frac{\Gamma \vdash g : \gamma_d \rightarrow \{\mathsf{gen} : \Sigma \rightarrow \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\uparrow}_{m_k} \rightarrow \tau^{\downarrow}_{m_k}\}^{j \in New, k \in Redef}, \mathsf{superinit} : \gamma_b\}}{\Gamma \vdash \mathsf{mixinval}\langle g, \textit{New}, \textit{Redef}, \textit{Expect}\rangle : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle}\ (\textit{mixin val})$$

$$\text{where}\quad \begin{array}{l} \Sigma = \Sigma_{new} \cup \Sigma_{red} \cup \Sigma_{exp} \\ \Sigma_{new} = \{m_j : \tau^{\downarrow}_{m_j}\},\ \Sigma_{red} = \{m_k : \tau^{\downarrow}_{m_k}\},\ \Sigma_{exp} = \{m_i : \tau^{\uparrow}_{m_i}\},\ \Sigma_{old} = \{m_k : \tau^{\uparrow}_{m_k}\} \\ \tau^{\uparrow}_{m_i} \text{ and } \tau^{\uparrow}_{m_k} \text{ are inferred from method bodies and } i \in \textit{Expect} \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old}\rangle \\ \Gamma \vdash e_2 : \mathsf{class}\langle \gamma_c, \Sigma_b\rangle \\ \Gamma \vdash \gamma_b <: \gamma_c \\ \Gamma \vdash \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{old}) \\ \Gamma \vdash \Sigma_{red} <: \Sigma_b / \Sigma_{red} \\ \textit{Subj}(\Sigma_b) \cap \textit{Subj}(\Sigma_{new}) = \emptyset \end{array}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{class}\langle \gamma_d, \Sigma_d\rangle}\ (\textit{mixin app})$$

$$\text{where}\quad \Sigma_d = \Sigma_{new} \cup \Sigma_{red} \cup (\Sigma_b - \Sigma_b / \Sigma_{red})$$

Fig. 5. Typing rules for class and mixin-related forms

argument of type γ_{b_1} to the constructor of the superclass e_2, that expects an argument of type γ_{d_2} for its constructor. Therefore, we require that $\gamma_{b_1} <: \gamma_{d_2}$ (condition (c_1)).

The mixin e_1 is allowed to redefine methods: defined by e_2, expected by e_2, or redefined by e_2. In all cases we must check that the redefinition (and the expectation about the old method in the superclass) is type safe (conditions (c_2), (c_3) and (c_4)). If e_1 redefines a method m_k that is in turn redefined by e_2, then we will put the redefined type of m_k from e_1 in Σ_{red} and the old one from e_2 in Σ_{old}. This is consistent with the view that the new mixin will contain m_k with the body from e_1 (with its *next* bound to e_2's implementation, while in m_k's body from e_2 *next* remains still unbound, as the method m_k can be further redefined, see Section 3). If e_1 redefines, instead, an expected method of e_2, that method will not appear in Σ_{exp}, but the redefined type and the old type, as inferred from e_1, will appear in Σ_{red} and Σ_{old}, respectively.

$$\dfrac{\begin{array}{l} \quad\;\; \Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_{b_1}, \gamma_{d_1}, \Sigma^1_{new}, \Sigma^1_{red}, \Sigma^1_{exp}, \Sigma^1_{old} \rangle \\ \quad\;\; \Gamma \vdash e_2 : \mathsf{mixin}\langle \gamma_{b_2}, \gamma_{d_2}, \Sigma^2_{new}, \Sigma^2_{red}, \Sigma^2_{exp}, \Sigma^2_{old} \rangle \\ (c_1)\ \Gamma \vdash \gamma_{b_1} <: \gamma_{d_2} \\ (c_2)\ \Gamma \vdash \tau^{\downarrow}_{m_{k_1}} <: \tau'^{\downarrow}_{m_{j_2}} <: \tau^{\uparrow}_{m_{k_1}} \quad \text{if } k_1 = j_2 \\ (c_3)\ \Gamma \vdash \tau^{\downarrow}_{m_{k_1}} <: \tau'^{\downarrow}_{m_{k_2}} <: \tau^{\uparrow}_{m_{k_1}} \quad \text{if } k_1 = k_2 \\ (c_4)\ \Gamma \vdash \tau^{\downarrow}_{m_{k_1}} <: \tau'^{\uparrow}_{m_{i_2}} <: \tau^{\uparrow}_{m_{k_1}} \quad \text{if } k_1 = i_2 \\ (c_5)\ \Gamma \vdash \tau'^{\downarrow}_{m_{j_2}} <: \tau^{\uparrow}_{m_{i_1}} \quad \text{if } i_1 = j_2 \\ (c_6)\ \Gamma \vdash \tau'^{\downarrow}_{m_{k_2}} <: \tau^{\uparrow}_{m_{i_1}} \quad \text{if } i_1 = k_2 \\ (c_7)\ \Gamma \vdash \tau'^{\uparrow}_{m_{i_2}} <: \tau^{\uparrow}_{m_{i_1}} \vee \Gamma \vdash \tau^{\uparrow}_{m_{i_1}} <: \tau'^{\uparrow}_{m_{i_2}} \quad \text{if } i_1 = i_2 \\ (c_8)\ \mathit{Subj}(\Sigma^1_{new}) \cap (\mathit{Subj}(\Sigma^2_{new}) \cup \mathit{Subj}(\Sigma^2_{red}) \cup \mathit{Subj}(\Sigma^2_{exp})) = \emptyset \end{array}}{\Gamma \vdash e_1 \bullet e_2 : \mathsf{mixin}\langle \gamma_{b_2}, \gamma_{d_1}, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle}\ (\mathit{mixin\ comp})$$

$$\text{where} \quad \begin{array}{c} \Sigma^1_{new} = \{m_{j_1} : \tau^{\downarrow}_{m_{j_1}}\}, \Sigma^2_{new} = \{m_{j_2} : \tau'^{\downarrow}_{m_{j_2}}\}, \Sigma^1_{red} = \{m_{k_1} : \tau^{\downarrow}_{m_{k_1}}\}, \Sigma^2_{red} = \{m_{k_2} : \tau'^{\downarrow}_{m_{k_2}}\} \\ \Sigma^1_{exp} = \{m_{i_1} : \tau^{\uparrow}_{m_{i_1}}\}, \Sigma^2_{exp} = \{m_{i_2} : \tau'^{\uparrow}_{m_{i_2}}\}, \Sigma^1_{old} = \{m_{k_1} : \tau^{\uparrow}_{m_{k_1}}\}, \Sigma^2_{old} = \{m_{k_2} : \tau'^{\uparrow}_{m_{k_2}}\} \end{array}$$

$$\begin{array}{l} \Sigma_{new} = \Sigma^1_{new} \cup (\Sigma^2_{new} - \Sigma^2_{new}/\Sigma^1_{red}) \cup \Sigma^1_{red}/\Sigma^2_{new} \\ \Sigma_{red} = (\Sigma^1_{red} - \Sigma^1_{red}/\Sigma^2_{new}) \cup (\Sigma^2_{red} - \Sigma^2_{red}/\Sigma^1_{red}) \\ \Sigma_{old} = (\Sigma^1_{old} - (\Sigma^1_{old}/\Sigma^2_{new} \cup \Sigma^1_{old}/\Sigma^2_{old})) \cup \Sigma^2_{old} \\ \Sigma_{exp} = (\Sigma^1_{exp} - (\Sigma^1_{exp}/\Sigma^2_{new} \cup \Sigma^1_{exp}/\Sigma^2_{red} \cup \Sigma^1_{exp}/\Sigma^2_{exp})) \cup (\Sigma^2_{exp} - (\Sigma^2_{exp}/\Sigma^1_{red} \cup \Sigma^2_{exp}/\Sigma^1_{exp})) \cup \Sigma_{min} \\ \Sigma_{min} = \{m_i : \min\{\tau^{\uparrow}_{m_i}, \tau'^{\uparrow}_{m_i}\} \mid m_i : \tau^{\uparrow}_{m_i} \in \Sigma^1_{exp}, m_i : \tau'^{\uparrow}_{m_i} \in \Sigma^2_{exp}\} \end{array}$$

Fig. 6. Typing rule for mixin composition

Conditions (c_5) and (c_6) check whether e_2 can provide methods (either defined or redefined) that are expected by e_1. If such a method is provided, then it will not appear in Σ_{exp}. In case both e_1 and e_2 expect the same method, the types with which such method is expected must be comparable (condition (c_7)); the method will then appear in Σ_{exp} with the smaller type.

Finally, condition (c_8) checks that no name clash occurs among methods defined by e_1 and those defined/redefined/expected by e_2. This decision is in line with a good object-oriented design principle of not confusing method redefinitions and name clashes.

Our system is proved sound, in the sense that "every well-typed program cannot go wrong", which implies the absence of *message-not-understood* runtime errors.

We consider *programs*, which are closed terms, and we introduce *faulty programs*, which are a way to approximate the concept of reaching a "stuck state" during the evaluation process. Intuitively, a program reaches a "stuck state" if, for instance, a method call is attempted on an expression that does not evaluate to an object. We prove that if the evaluation for a given program p does not diverge, then either p returns a value, or p reduces to a faulty program. We then show that faulty programs are not typeable, and, via a subject reduction property, we establish that if a program is typeable in our system, then it evaluates to a value, under the condition that the program does not diverge.

Lemma 1 (Subject Reduction). *If $\Gamma \vdash e : \tau$ and e evaluates to e', then $\Gamma \vdash e' : \tau$.*

Theorem 1 (Soundness). *Let p be a program: if $\varepsilon \vdash p : \tau$ then either the evaluation for p diverges, or p evaluates to a value v and $\varepsilon \vdash v : \tau$ (ε stands for the empty typing environment).*

The metatheory for the present system, and in particular the subject reduction property, are extensions of the ones in [9] (Chapter 9). The formal definitions and properties were analyzed in detail, and can be found at:
`http://www.dsi.unifi.it/~bettini/high-proofs.pdf`.

5 Conclusions

This paper presents a calculus supporting class hierarchies creation via mixin application (already present in [10]) and mixin composition. Our goal was to design a clean and general form of mixin composition without committing ourselves to an already existing language. We chose to extend the calculus [10] because: (*i*) it is an easy-to-extend framework; (*ii*) its operational semantics is close both to an implementation and to a denotational model. Therefore, being able to produce something towards a denotational model for mixins is, in our opinion, a good by-product; (*iii*) it allowed us to choose *structural subtyping* (as opposed to *nominal subtyping* of C++ and Java), since, according to Bracha et al., [5], "When subtyping is structural, mixins do not introduce any new issues with respect to subtyping." Moreover, it has the advantage of being independent from the class hierarchy.

In the literature, there are many proposals that deal with mixins. We mention here some of them, the most interesting with respect to our calculus. Bracha and Cook extend Modula-3 with mixins in [14] (this is one of the seminal papers on mixins). The novelty is in seeing object types as mixins, which either explicitly state the modifications to the superclass, or are obtained as a result of mixin composition. The left-hand mixin has a "priority" and the composition is not explicitly written in order to ensure upward compatibility with the existing language. Instead, we think that making the composition explicit (as it is in our calculus) makes the programmer aware of how software components are composed, thus providing more control over the behavior of the program.

Flatt et al. [20] extend a subset of sequential Java called CLASSICJAVA with mixins and call it MIXEDJAVA. Mixins use their *inheritance interface* to specify how the inherited methods are extended and/or overridden. Existing mixins can be combined in order to produce new composite mixins. As in our calculus, the left-hand mixin has the "precedence" over the right-hand mixin. Composition is well-defined only if the right-hand mixin implements the left-hand mixin inheritance interface (i.e., the right-hand mixin is required to provide all the methods expected by the left-hand one). In this respect, our approach is more oriented to code composition, in that the new composite mixin is still allowed to have expected methods not yet resolved. The duplication of method names in MIXEDJAVA is resolved at run-time with the run-time context information provided by the current *view* of the object (represented as a chain of mixins).

Ancona and Zucca [2–4] give a formal model for mixin modules. A mixin is a function from input to output components, and they characterize axiomatically the operators for composing mixins in order to obtain higher-order mixins. They also present a variety of method renaming forms, to deal with different typologies of name collisions. In [1] they present JAM, an extension of Java supporting mixins, but not mixin composition, where name collision is treated essentially as "accidental override".

Our approach is different from the ones of MIXEDJAVA and JAM in some respects. Besides not being a Java-like calculus, which allows us to use structural subtyping, our calculus has a more modular class constructor. Moreover, method names collisions are resolved statically by the type system. If this approach may look more restrictive than the ones of MIXEDJAVA and JAM, we preferred it because it forces the programmer to be aware of collisions and to resolve them, while automatic handling of such ambiguities may lead to unexpected behavior at run-time.

Boudol [12] extends *Reference* ML [25] with records and *let rec* operator. This enriched ML leads to a theoretically solid treatment of mixins, which are seen as class transformers. A marginal difference between the two calculi seems to be in the way references to fields are created. In our calculus these are created at class creation time, when mixin application is evaluated, whereas in the calculus of Boudol they are created at class instantiation time, i.e., when an object is created.

A future research direction is an extension of this calculus where not only classes can be instantiated but also mixins, obtaining a form of *incomplete objects*, to be completed in an object-based fashion. A first version of the incomplete objects is given in [6]. Moreover, higher-order mixins seem to be a natural feature to be added to MoMi [7], a coordination language where object-oriented mobile code is exchanged among the nodes of a network.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam—Designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, 2003.
2. D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *MSCS*, 8(4):401–446, 1998.
3. D. Ancona and E. Zucca. True modules for Java-like languages. In *Proc. of ECOOP 2001*, volume 2072 of *LNCS*, pages 354–380. Springer-Verlag, 2001.
4. D. Ancona and E. Zucca. A theory of mixin modules: Algebraic laws and reduction semantics. *MSCS*, 12, 2002.
5. L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. Mixins in Strongtalk. In *Proc. Inheritance Workshop at ECOOP 2002*, 2002.
6. L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Mixin-Based Incomplete Objects. In *Proc. of FOOL 11*, 2004.
7. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In *Proc. of Coordination Models and Languages*, volume 2315 of *LNCS*, pages 56–71. Springer, 2002.
8. 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.
9. V. Bono. *Type Systems for the Object Oriented Paradigm*. PhD thesis, Univ. di Torino, 1999.
10. V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *Proc. ECOOP '99*, pages 43–66. LNCS 1628, Springer-Verlag, 1999.
11. V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of classes and objects. In *Proc. 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS '99)*, volume 220, 1999.
12. G. Boudol. The recursive record semantics of objects revised. In *Proc. ESOP '01*, pages 269–283. LNCS 2028, Springer-Verlag, 2001.
13. N. Boyen, C. Lucas, and P. Steyaert. Generalized mixin-based inheritance to support multiple inheritance. Technical Report vub-prog-tr-94-12, Vrije Universiteit Brussel, 1994.
14. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA*, 1990.
15. K. Bruce. *Foundations of Object-Oriented Languages – Types and Semantics*. The MIT Press, 2002.

16. W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
17. E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. POPL '91*, pages 233–244, 1991.
18. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
19. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, 1998.
20. M. Flatt, S. Krishnamurthi, and M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, 1999.
21. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
22. M. V. Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple iinheritance problems. *Object Oriented Systems*, 3(1), 1996.
23. I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proc. ICALP '89*, pages 574–588. LNCS 372, Springer-Verlag, 1989.
24. A. Patel. *Obstacl: a language with objects, subtyping, and classes*. PhD thesis, Stanford University, 2001.
25. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Subtyping Rules and Other Typing Rules

$$\frac{}{\Gamma, \iota_1 <: \iota_2 \vdash \iota_1 <: \iota_2}\ (<:\ \mathit{proj}) \qquad \frac{}{\Gamma \vdash \tau <: \tau}\ (\mathit{refl}) \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}\ (\mathit{trans})$$

$$\frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash \tau \to \sigma <: \tau' \to \sigma'}\ (\mathit{arrow}) \qquad \frac{\Gamma \vdash \tau_i <: \sigma_i \ \ i \in I \quad I \subseteq J}{\Gamma \vdash \{m_i : \tau_i\}^{i \in I} <: \{m_j : \sigma_j\}^{j \in J}}\ (<:\ \mathit{record})$$

$$\frac{\mathit{typeof}(\mathit{const}) = \tau}{\Gamma \vdash \mathit{const} : \tau}\ (\mathit{const}) \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\mathit{proj}) \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \to \sigma}\ (\lambda) \qquad \frac{}{\Gamma \vdash\ ! : \tau\ \mathsf{ref} \to \tau}\ (!)$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}\ (\mathit{app}) \qquad \frac{}{\Gamma \vdash \mathit{fix} : (\sigma \to \sigma) \to \sigma}\ (\mathit{fix}) \qquad \frac{\Gamma \vdash e : \{x : \sigma\}}{\Gamma \vdash e.x : \sigma}\ (\mathit{lookup})$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau <: \sigma}{\Gamma \vdash e : \sigma}\ (\mathit{sub}) \qquad \frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{x_i = e_i\}^{i \in I} : \{x_i : \tau_i\}}\ (\mathit{record}) \qquad \frac{}{\Gamma \vdash \mathsf{ref} : \tau \to \tau\ \mathsf{ref}}\ (\mathit{ref})$$

$$\frac{}{\Gamma \vdash :=\ \ : \tau\ \mathsf{ref} \to \tau \to \tau}\ (:=) \qquad \frac{\Gamma' = \Gamma, x_1 : \tau_1\ \mathsf{ref}, \ldots, x_n : \tau_n\ \mathsf{ref} \quad \Gamma' \vdash v_i : \tau_i \quad \Gamma' \vdash e : \tau}{\Gamma \vdash H\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : \tau}\ (\mathit{heap})$$