A Core Calculus of Mixin-Based Incomplete Objects (extended abstract)*

Lorenzo Bettini[1] Viviana Bono[2] Silvia Likavec[2]
[1]Dip. di Sistemi ed Informatica, Univ. di Firenze, Via C. Lombroso 6/17, 50134 Firenze, Italy,
bettini@dsi.unifi.it
[2]Dip. di Informatica, Univ. di Torino, C.so Svizzera 185, 10149 Torino, Italy, {bono,likavec}@di.unito.it

Abstract. We design a calculus that combines class-based features with object-based ones, with the aim of fitting into a unifying setting the "best of both worlds". In a mixin-based approach, mixins are seen as *incomplete classes* from which *incomplete objects* can be instantiated. In turn, incomplete objects can be completed in an object-based fashion. Our hybrid calculus is shown to be useful in some real world scenarios that we present as examples.

1 Introduction

In object-oriented *class-based* languages, objects (as fully-fledged instances of classes) are the computational entities of a software system, but they are "passive" with respect to their structure, which is henceforth fixed by a class hierarchy. In object-oriented *object-based* languages, objects are the computational entities and, at the same time, they govern the inheritance mechanism (through operations like method addition and method override, that produce new objects starting from the existing ones). The class-based paradigm is the one of production languages (like Java and C++), mainly because it helps imposing a programming discipline that is necessary especially in a programming-in-the-large setting. The object-based paradigm is gaining a renewed interest in the context of script languages, like Java-Script (see for example [10, 22]), and delegation-based languages (see for example [4, 5]), where programming should produce "plug-and-play" prototypes easy and quick to implement also for non-expert programmers. For some insightful discussions about the differences between the two paradigms observed from different angles, we refer the reader to the books of Abadi-Cardelli [1] and of Bruce [18].

Our aim is to design a calculus that combines class-based features with object-based ones, that is, to try to fit into one setting the "best of both worlds", discipline and flexibility first of all. Object-based objects are prone to be transformed at any point in a program, because they can generate other objects with more/different functionalities via method addition and override. Mixins can be seen as *incomplete* classes, and their instances would be *incomplete* objects that could be completed in an object-based fashion. Therefore, we think that the best suited inheritance mechanism to be integrated with the object-based paradigm is a *mixin*-based one more than a pure class-based one. Hence, in our calculus it will be possible: (*i*) to instantiate classes (created via mixin-based inheritance), obtaining fully-fledged objects ready to be used; (*ii*) to instantiate *mixins*, yielding *incomplete objects* that can be partly usable but they might still need to be completed in an object-based fashion (via *method addition* and/or *object composition*). In other words, it will be possible to design tidy yet flexible hierarchies of classes through mixin application to be used in the classical way, but also to experiment with prototypical incomplete objects. Section 3 will provide some scenarios in which it looks desirable having such a hybrid calculus: in object-oriented design there are some situations when one would like to add simply new features to existing objects by means of functions, without having to resort to writing new classes and mixins for this purpose only.

Our proposal is a formal calculus equipped with an operational semantics and a type system to capture statically "message-not-understood" run-time errors, but the aim of this paper is mainly to introduce the reader to the design of this hybrid "mixin-object-based calculus". Therefore, the paper can be seen as divided into two parts: the first one, quite self-contained, is devoted to an informal yet rather complete description of the proposal, and the second one presents its formal description via the operational semantics. In this extended abstract the type system is omitted, and can be found with the metatheory at `http://www.dsi.unifi.it/~bettini/incproofs.ps`. In particular, we proved a *soundness* property,

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

stating that "well-typed computations cannot go wrong", i.e., that we avoid message-not-understood run-time errors.

The paper is organized as follows. In Section 2 we describe the most prominent features of our calculus and its syntax. In Section 3 we present some motivating scenarios for our proposal. Section 4 introduces the operational semantics of the calculus. We conclude the paper with some comparison with related work and future research directions.

2 The mixin calculus

Recently, the concept of "mixin" is undergoing a renaissance, both as a mixin "class" (parametrized (sub)class definition [2, 7, 8]), and as a mixin "module" (module supporting deffered components [3, 24, 30]), due to their flexible nature of "incomplete" entities prone to be completed according to the programmer's needs. In this paper the term *mixin* refers to *mixin classes*.

Mixins [17, 28] are (sub)class definitions parameterized over a superclass and were introduced as an alternative to some forms of multiple inheritance [15, 16, 31]. A mixin could be seen as a function that, given one class as an argument, produces another class, by adding and/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. The superclass definition is not needed at the time of writing the mixin definition. This minimizes the dependencies between superclass and its subclasses, as well as between class implementors and end-users, thus improving modularity and code reuse.

In this work we extend the core calculus of classes and mixins of [12] with *incomplete objects*: a mixin can (*i*) be applied to a class to create a fully-fledged subclass; or (*ii*) be instantiated to obtain an incomplete object. In turn, an incomplete object can be completed via *method addition* (this operation is discussed to greater extent in [26] and in [25]) or via *object composition*.

A mixin contains three sorts of method declarations: *new* methods, which are the newly introduced methods by the mixin seen as a subclass, *redefining* methods, which wait for a superclass containing a method with the same name to be redefined, and provide the overriding body, and *expected* method names, which are methods not implemented by the mixin but must be provided by the superclass (because they might be used by the new and redefining methods). A different choice would be to declare only expected and new methods, inferring the redefining ones by intersecting the former two, but this choice would not change much the semantics of the calculus, while making the syntax more difficult to use by a programmer. Redefining methods can refer to their corresponding superclass implementation via a special keyword *next*. We remark that we choose to refer to the corresponding super-method only, instead of permitting the access to all of the superclass' methods (as **super** in Java does), because this extension is just a technical matter and including it would complicate the operational semantics without adding anything relevant to the calculus. We assume that the programmer must declare the expected method names in the mixins, but that their types are inferred from new and redefining method bodies[1]. A mixin type takes into account the types of the methods already present in the mixin and the expected types of the components not yet present, i.e., the names and the types of the expected methods and the types of the "super-class" method bodies (*next*s) of the redefining methods.

If we apply a mixin to a class via the *mixin application* operation, we obtain a new fully-fledged subclass. In order to have a successful mixin application, the (super)class must provide everything that is listed in the mixin information; in particular, the types of the methods provided by the class must be (according to a standard subtyping relation) "equal or better" than the ones required by the mixin, with respect to the expected methods, and "equal or worse" if they are overridden.

When we instantiate a mixin we obtain an incomplete object that can be "incomplete" in two different respects: (*i*) it may need some expected methods; (*ii*) it may contain redefining methods that cannot be invoked until methods with the functionality of their *next* is added. Method invocation is allowed on incomplete objects, but only methods that are fully complete, i.e., the ones that do not need a *next* and do not use either expected or incomplete methods, can be invoked. An appropriate type rule forbids non-correct method invocations on incomplete objects.

[1] Expected method names could also be inferred, but we decided to have them declared to simplify the description of the operational semantics. Also inferring the expected types is a choice that can be changed according to the features of the language we might want to enrich with the mixin machinery, for instance, inference would not be easy in presence of overloading.

Completion can happen in two ways: (*i*) via *method addition*, that can add one of the expected methods or one of the missing *next*s; (*ii*) via *object composition*, that takes an incomplete object and composes it with a complete one containing all the required methods. The incomplete object has a type that carries constraint information like the type of the mixin it has been instantiated from. Thus, method addition and object composition must respect the same constraints as the mixin application. The type system ensures that all method additions and object compositions are type safe and that only "complete" methods are invoked on objects. Furthermore, method addition can only act on incomplete objects, and the object composition completes an incomplete object with a complete one. This way we totally exploit the type information at the mixin level, obtaining a "tamed" and safe object-based calculus at the object level. General method addition and a form of object-based override are under study as extensions of our calculus, but in this paper we want to focus on the design features and the uses of the "tamed" method addition only.

When a method is added, it becomes an effective component of the host object, i.e., not only the methods of the host object may invoke it, but also the new added method can use any of its sibling methods. This is rendered by requiring that all methods, hence also the ones which are added through method addition or object composition, must be a function of *self*. In this way, the reference to the host object can be updated every time a method addition or an object composition takes place, in order to take into consideration the new methods. This automatically enables correct dynamic binding for all methods, i.e., if for some of the methods their corresponding *next* methods are provided via addition or composition, the redefined versions of such methods will be dynamically invoked by the other methods. As explained in Section 3, this ensures a real *delegation* mechanism in object composition.

As in the calculus of [12], our calculus is imperative and does not support *MyType* [26] inheritance (and as such does not support *binary methods* [19]). In this first version of the calculus we assume that the methods we add to an incomplete object via addition or composition do not introduce incompleteness themselves, i.e., the set of "non-ready" methods never increases. Moreover, we do not consider *higher-order* mixins (mixins that can also be applied to other mixins yielding other mixins) and related mixin composition, being an orthogonal issue. We would like to point out, though, that our study of higher-order mixins (which can be found in [6]) helped us finding the right formalization for incomplete objects.

2.1 Design choices

Our formal design choices are strongly based on the ideas presented in [12, 33]. The leading idea is that modular program development in a class-based language involves minimizing code dependencies not only between a superclass and its subclasses, as obtained by using mixin-based inheritance, but also between class implementation and object users.

As some other theoretical calculi for object-oriented languages, our calculus directly supports modular object construction ([11, 25, 27, 34, 35]). The mixin implementor only writes the local constructor for his own mixin. Mixin applications are reduced to generator functions which call all the constructors in the inheritance chain in correct order, producing a fully initialized object (see Section 4).

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[2]. 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.

To ensure that mixin inheritance can be statically type checked, the calculus employs subtype-constrained parameterization. From each mixin definition the type system infers a constraint specifying to which classes the mixin may be applied so that the resulting subclass is type-safe. The constraint includes 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 ("behavior" being formalized via types). Having this distinction in the syntax of our calculus helps mixin implementors avoid unintentional redefinitions of superclass methods.

[2] The reader will see how our operational semantics really implements this feature, more than merely assuming it as a general property for private fields.

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

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

This paper deals with incomplete objects remaining faithful to the above described principles. Meaningful differences and extensions to the original syntax of [12] are commented below.

2.2 Syntax of the calculus

The calculus of classes and mixins of [12] is based on *Reference ML* of Wright and Felleisen [36]. To the imperative calculus of records, functions, classes and mixins of [12] we add the machinery to work with incomplete objects. The class and mixin related expressions are: classval (class value), mixin, $\diamond$ (mixin application), and new. The (in)complete-object related expressions are:

- $\mathsf{mixinval}\langle v_g, New, Redef, Expect\rangle$ (mixin value),
- $\mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in I}, v_g\rangle$ (complete object),
- $\mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in I}, v_g, r, New, Redef, Expect\rangle$ (incomplete object),
- $\longleftarrow$ (method addition/object composition).

Mixins are first class citizens in our calculus, which means that all the usual operations can be performed on them (they can be passed as arguments to functions, be returned as results to function calls or combined in order to get new data structures). However, class values, mixin values, and object forms are not intended to be written directly; instead, these expression forms are used only to define the semantics of programs. Class values can be created by mixin application, mixin values result from evaluation of mixins, and object forms can be created by class and mixin instantiation.

The lambda-calculus related forms in Figure 1 are standard. We describe below the other forms.

- ref, !, := are operators[3] for defining a reference to a value, for dereferencing 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.
- 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 also have a concept of *store* or *heap*, represented by h in the expression $\mathsf{H}h.e$, which is 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.
- $\mathsf{classval}\langle v_g, \mathcal{M}\rangle$ 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 the index, i.e., $m_i = m_j$ if and only if $i = j$, so method names are identified with their indices.
- 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*, and it explicitly states the methods that are newly introduced by the mixin (m_j), redefined in the mixin (m_k), and expected by the mixin from the superclass (m_i). Each method body $v_{m_{j,k}}$ is a function

[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 [36], this is just a syntactic convenience, as is the curried version of :=.

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 old, redefined method from the superclass. The v_c value in the constructor clause is a function of an argument 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 4. The generator basically binds the private field in all method bodies.

- new e uses generator v_g of the class value or of the mixin value to which e evaluates to create a function that returns a new object (incomplete, in the mixin case), as described in Section 4.
- $e_1 \diamond e_2$ denotes the application of mixin e_1 to class value e_2, producing a new class value which is a subclass of e_2 (mixin application is considered to be the basic inheritance mechanism in our calculus).
- $e_1 \longleftarrow m_i = e_2$ is the method addition operation: it adds the definition for method m_i with body e_2 to the (incomplete) object to which e_1 evaluates.
- $e_1 \longleftarrow e_2$ is the object composition operation: it composes the (incomplete) object to which e_1 evaluates with the complete object to which e_2 evaluates.
- $\mathsf{mixinval}\langle v_g, New, Redef, Expect\rangle$ is a *mixin value*, and it is the result of mixin evaluation. The generator v_g for the mixin is a "partial generator", that is, a generator of incomplete objects. This partial generator is used also in the $\diamond$ operation evaluation, where it is appropriately composed with the class generator (see the operational semantics in Section 4). The sets *New*, *Redef* and *Expect* contain the indices of new, redefining and expected methods defined in the mixin.
- $\mathsf{obj}\langle\{m_i = v_{m_i}\}^{i\in I}, v_g\rangle$ is a fully-fledged object that might have been created by directly instantiating a class, or by completing an incomplete object. Its first component is a record of methods, the second component is a generator function, which is kept also for complete objects, since they can be used to complete the incomplete ones.
- $\mathsf{obj}\langle\{m_i = v_{m_i}\}^{i\in I}, v_g, r, New, Redef, Expect\rangle$ is an incomplete object. It contains a record of methods $\{m_i = v_{m_i}\}^{i\in I}$, a generator function v_g, a record r containing redefining methods which will be used when a *next* for them becomes available during method addition or object composition (as explained in Section 4), and three sets *New*, *Redef*, and *Expect*, containing the indices of new, redefining and expected methods defined in the mixin. When the sets of method names *Redef* and *Expect* become empty (and so does the record of redefining methods) the incomplete object becomes a complete object.

Finally, we define the root of the class hierarchy, class *Object*, as a predefined class value:

$$Object \triangleq \mathsf{classval}\langle\ \lambda\_.\lambda\_.\{\},\ [\ ]\ \rangle$$

The root class is necessary so that all other classes can be treated uniformly. 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*.

With respect to the calculus of [12], we introduced *mixin values* (that we studied in the context of higher-order mixins, see [6]). They are partially evaluated incomplete classes (as it can be seen in Section 4), which are quite handy to use when creating incomplete objects. It is important to notice that they bear exactly the same types of the mixins they are evaluated from, and this points out that they are more syntactic sugar than substantial syntax, which becomes useful when defining the operational semantics. Moreover, in the original calculus, objects were simply records, while in this paper they are tuples, in order to be able to deal with incompleteness. Still, invocable methods are all contained in a record-shaped component of the tuple.

3 Examples

In this section, we provide some real life pragmatic examples that demonstrate how incomplete objects and object completion via method addition and object composition can be used fruitfully to design complex systems, since they supply programming tools that may make software development easier.

For readability, we will use here a slightly simplified syntax with respect to the calculus presented in Section 2.2: (*i*) we will list the methods' parameters in between "()"; (*ii*) $e_1; e_2$ is interpreted as $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\ x \notin FV(e_2)$, coherently with a call-by-value semantics; (*iii*) we will avoid to make references explicit, thus $\mathsf{let}\ x = e\ \mathsf{in}\ x.m()$ should be intended as $\mathsf{let}\ x = \mathsf{ref} e\ \mathsf{in}\ (!x).m()$; (*iv*) method bodies are only sketched.

3.1 Object completion via method addition

In the first example, we present a scenario where it is suitable to add some functionalities to existing objects without writing new mixins and creating related classes only for this purpose. Let us consider the develop-

```
let Button =
 mixin
  method display = ...
  method setEnabled = ...
  expect onClick;
  ...
 end in

let MenuItem =
 mixin
  method show = ...
  method setEnabled = ...
  expect onClick;
  ...
 end in

let ShortCut =
 mixin
  method setEnabled = ...
  expect onClick;
  ...
 end in

let ClickHandler =
 (λ doc. λ self. ... doc.save() ... self.setEnabled(false)) mydoc
 in
  let button = new Button("Save") in
  let item = new MenuItem("Save") in
  let short = new ShortCut("Ctrl+S") in
   button.display();
   button ⟵+ (OnClick = ClickHandler);
   button.setEnabled(true);
   mydialog.addButton(button); // now it is safe to use it
   item ⟵+ (OnClick = ClickHandler);
   item.setEnabled(true);
   mymenu.addItem(item);
   short ⟵+ (OnClick = ClickHandler);
   short.setEnabled(true);
   system.addShortCut(short);
```

Fig. 2. Widget example

ment of an application that makes use of widgets such as graphical buttons, menus and keyboard shortcuts. These widgets are usually associated to an event listener (e.g., a callback function), that is invoked when the user sends an event to that specific widget (e.g., one clicks the button with the mouse or chooses a menu item).

The design pattern *command* [29] is very useful for implementing these scenarios, in that it allows parameterization of the widgets over the event handlers, and the same event handler can be reused for similar widgets (e.g., the handler for the event "save file" can be associated with a button, a menu item, or a keyboard shortcut). However, in such a context, it is more convenient to be able to simply add a function without creating a new mixin just for this aim. Indeed, the above mentioned pattern seems to provide a solution in pure class-based languages that normally do not supply the object method addition operation.

Within our approach, this problem can be solved with language constructs: mixin instantiation (to obtain an incomplete object which can be seen as a prototype) and method addition/completion (in order to provide the further functionalities needed by the prototype). For instance, we could implement the solution as in Figure 2. The mixin `Button` expects (i.e., uses but does not implement) a method `onClick` that is internally called when the user clicks on the button (e.g., by the window where it is inserted, in our example the dialog `mydialog`). When instantiated, it creates an incomplete object that can be used for invoking methods that are already usable (e.g., `display`, provided it does not use `onClick`). Then the `button` object is completed with the event listener `ClickHandler` (by using method addition). This listener is a function that has the parameter `doc` already bound to the application main document. Once the object is completed it can be safely enabled. Notice that the added method can rely on methods of the host object (e.g., `setEnabled`). The same listener can be installed (by using method addition again) to other incomplete objects, e.g., the menu item `"Save"` and the keyboard shortcut for saving functionalities. Moreover, since we are able to act directly on instances here, our proposal also enables customization of objects at run-time.

The following piece of code (that works together with the previous one) shows another example of object completion via *method addition*, where the method to be completed expects the implementation of the superclass (it refers to it via *next*):

```
let FunnyButton =
 mixin
  method display = ...
  method setEnabled = ...
  method playSound = ...
  redefine onClick = λself. λnext. ... next() ...
   self. playSound("tada.wav");
 end in

let funnybutton = new FunnyButton("Save") in
 funnybutton.display();
 funnybutton ⟵+ (OnClick = ClickHandler);
 funnybutton.setEnabled(true);
 // now it is safe to use it
 toolbar.addButton(funnybutton);
```

```
let File =
  mixin
   method write = ...
   method read = ...
   ...
  end in

let Socket =
  mixin
   method write = ...
   method read = ...
   method IP = ...
   ...
  end in

let Console =
  mixin
   method write = ...
   method read = ...
   method setFont = ...
   ...
  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 ...
let Buffer =
  mixin
   redefine write = λ size. λ self. λ next. λ data.  // bufferize write requests;
   redefine read = λ size. λ self. λ next. λ _ .  // read from the buffer;
   constructor λ (size, arg). {fieldinit=size, superinit=arg};
  end in ...
```

Fig. 3. Stream example

In fact, the mixin `FunnyButton` does not simply expect the method `onClick`, it expects to redefine such method: the redefined method relies on the implementation provided by *next* method (either provided by a superclass, or in this example directly added via method addition to an object instance of `FunnyButton`) and adds a "sound" to the previous implementation. Notice that once again the previous event handler can be reused in this context, too.

3.2 Object completion by object composition

Object composition is often advocated as a powerful alternative to class inheritance in that it is defined at run-time and it enables dynamic object code reuse and composition by assembling existing components. Object composition is often used in conjunction with *delegation*: a receiving object delegates request handling to another object. However, this mechanism must be programmed explicitly. Furthermore, object composition is often the right flexible alternative to inheritance when functionalities have to be added dynamically to existing objects at run-time. These situations are usually dealt with by the pattern *decorator* [29]. However, also in the case of this pattern, explicit programming is required.

With our linguistic constructs for object completion, both object composition and delegation are automatically handled by the language. Indeed, the decorator pattern is easily implementable with these constructs. In this section we show how to exploit these features for implementing a logging system based on streams. Notice that streams are often implemented according to the decorator pattern.

In Figure 3 there are the definitions of `Compress` mixin and `Buffer` mixin that, respectively, implement compression and buffering functionality on top of any stream class. `File`, `Socket` and `Console` represent basic stream functionalities (for I/O on a file, on the net and on the standard input output, respectively). Note that the class to which the mixin is applied may have more methods than expected by the mixin. For example, `Compress` can be applied to `Socket` $\diamond$ *Object* even though `Socket` $\diamond$ *Object* has other methods besides *read* and *write*. Streams are created by composing streams with advanced functionalities like `Compress`, that are incomplete objects (i.e., instances of the appropriate mixin), with streams with basic functionalities like `File`:

```
let fileoutput =
 (new Compress("HIGH")) ⟷+(new (File ◇ Object) ("foo.txt"))  in
  fileoutput.write("bar")
```

The power of object composition can be seen when we compose more than one stream in a chain of objects. For example:

```
let fileoutput =
 (new UUEncode("base64")) ⟷+(new Compress("HIGH")) ⟷+(new Buffer(1024)) ⟷+
  (new (File ◇ Object) ("foo.txt"))  in
   fileoutput.write("bar")
```

Construction of decorations can be delegated to different parts of the program (thus exploiting modular programming) and the resulting incomplete objects can then be assembled in order to build a complete object. For instance, the following code delegates the construction of decorations for buffering and compression to two functions, and then assembles the returned objects and completes them:

```
let o1 = build_compression() in
 let o2 = build_buffering() in
  let out = o1 ⟷+ o2 ⟷+ (new(File ◇ Object)("foo.txt")) in
   out.write("bar")
```

The function `build_compression` returns a specific incomplete object according, e.g., to user's requests: it can return a simple `Compress` object, or an `UUEncode` one. Similarly, `build_buffering` takes care of building a buffering object. The two returned objects can be then completed with a chain of $\longleftrightarrow+$ operations.

Now we can program our logging functionalities exploiting the stream system shown above:

```
let Logger =
 mixin
  method doLog = λ verb. λ self. λ msg.
   write(self.getTime() + ":  " + msg);
  method getTime = ...
  expect write;
  ...
 end in
```

```
let logger = new Logger(verbosity) ⟷+ output in
 output.doLog("logging started...");
 output.doLog("log some actions...");
```

The `output` object can be any stream object we showed above. Indeed, it does not have to be a stream: it is only requested to provide the method `write`. This allows to build a more complex logging system by assembling more powerful components. For instance we can program a *multiplexer* that writes to many targets and use this multiplexer to complete our logger:

```
let Multiplexer =
 mixin
  method addTarget =
   // add the target to the list;
  method removeTarget =
   // remove the target from the list;
  method write =
   // call "write" on every object in the list
  ...
 end in
```

```
let multi = new Multiplexer ◇ Object in
 multi.addTarget((new Compress("HIGH")) ⟷+
  (new (File ◇ Object) ("foo.txt")));
 multi.addTarget((new Buffer(1024)) ⟷+
  (new (Socket ◇ Object) ("www.foo.it:9999")));
 let logger = new Logger(verbosity) ⟷+ multi in
  output.doLog("logging started...");
  output.doLog("log some actions...");
```

Notice that no explicit programming is required in order to structure classes for object composition and the presented form of method delegation: the programmer can simply concentrate on assembling the components as she likes. Furthermore, the type system will ensure that all object compositions are type safe and that only "complete" methods are invoked on objects.

4 Operational semantics

Our approach is the one of giving the calculus a semantics as close as possible to an implementation. In order to do so, the formal operational semantics is a set of rewriting rules including some standard rules for a lambda calculus with stores (in our case the Reference ML of Wright and Felleisen [36]), and some rules that evaluate the object-oriented related forms to records and functions, according to the object-as-record approach and Cook's class-as-generator-of-object principle. This operational semantics can be seen as something close to a denotational description for objects, classes, and mixins, and this "identification" of implementation and semantical denotation is, in our opinion, a good by-product of our approach.

The operational semantics extends the one of the core calculus of classes and mixins [12], therefore exploits the *Reference ML* of Wright and Felleisen [36] treatment of side-effects. 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 4, *R*'s are *reduction contexts* [21, 23, 32]. Reduction contexts are necessary to provide a minimal relative linear order among the creation, dereferencing and updating of heap locations, since side effects

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

Fig. 4. Reduction rules for standard expressions and heap expressions

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

Fig. 5. Reduction contexts

$$
\left(\begin{array}{l}
\mathsf{mixin} \\
\ \mathsf{method}\ m_j = v_{m_j}; \\
\ \mathsf{redefine}\ m_k = v_{m_k}; \\
\ \mathsf{expect}\ m_i; \\
\ \mathsf{constructor}\ c; \\
\mathsf{end}
\end{array}\right)^{\substack{j \in New \\ k \in Redef \\ i \in Expect}}
\to \mathsf{mixinval}\langle Gen_m, New, Redef, Expect\rangle \qquad (\mathrm{mixin})
$$

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

$$
\mathsf{new\ classval}\langle g, \mathcal{M}\rangle \to \lambda v.\mathsf{obj}\langle \mathit{fix}(g\ v), (g\ v)\rangle \qquad (\mathrm{new\ class})
$$

$$
\begin{array}{l}
\mathsf{new\ mixinval}\langle Gen_m, New, Redef, Expect\rangle \to \qquad (\mathrm{new\ mixin}) \\
\quad \lambda v.\mathsf{let}\ g = (Gen_m\ v)\ \ \mathsf{in} \\
\qquad \mathsf{obj}\langle \mathit{fix}(g.\mathsf{gen}), g.\mathsf{gen}, g.\mathsf{redef}, New, Redef, Expect\rangle
\end{array}
$$

$$
\mathsf{obj}\langle\{\ldots, m_i = v_{m_i}, \ldots\}, \_\rangle.m_i \to v_{m_i} \qquad (\mathrm{obj\ sel})
$$

$$
\mathsf{obj}\langle\{\ldots, m_i = v_{m_i}, \ldots\}, \_, \_, \_, \_, \_\rangle.m_i \to v_{m_i} \qquad (\mathrm{incobj\ sel})
$$

$$
\mathsf{mixinval}\langle Gen_m, New, Redef, Expect\rangle \diamond \mathsf{classval}\langle g, \mathcal{M}\rangle \to \mathsf{classval}\langle Gen, New \cup \mathcal{M}\rangle \qquad (\mathrm{mix\ app})
$$

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

Fig. 6. Reduction rules for object-oriented forms

need to be evaluated in a deterministic order. Their definition can be found in Figure 5. We assume the reader is familiar with the treatment of imperative side-effects via reduction contexts and we refer to [12, 36] for a description of the related rules.

The meaning of the object-oriented related rules in Figure 6 is as follows.

$$\mathsf{obj}\langle\{\ldots\},g,r,New,Redef,Expect\rangle \longleftarrow\!+\ (m_l = v_{m_l}) \rightarrow$$
$$\mathsf{let\ incgen} = \lambda self.$$
$$\left\{\begin{array}{ll} m_j = \lambda y.\ (g\ self).m_j\ y & {\scriptstyle j\in New} \\ m_k = \lambda y.\ self.m_k\ y & {\scriptstyle k\in Redef} \\ m_i = \lambda y.\ self.m_i\ y & {\scriptstyle i\in Expect-\{l\}} \\ m_l = \lambda y.\ v_{m_l}\ self\ y & \end{array}\right\} \qquad \text{(meth add 1)}$$
$$\mathsf{in\ obj}\langle \mathit{fix}(\mathrm{incgen}), \mathrm{incgen}, r, New\cup\{l\}, Redef, Expect-\{l\}\rangle$$

$$\mathsf{obj}\langle\{\ldots\},g,r,New,Redef,Expect\rangle \longleftarrow\!+\ (m_l = v_{m_l}) \rightarrow$$
$$\mathsf{let\ incgen} = \lambda self.$$
$$\left\{\begin{array}{ll} m_j = \lambda y.\ (g\ self).m_j\ y & {\scriptstyle j\in New} \\ m_k = \lambda y.\ self.m_k\ y & {\scriptstyle k\in Redef-\{l\}} \\ m_i = \lambda y.\ self.m_i\ y & {\scriptstyle i\in Expect} \\ m_l = \lambda y.\ (r.m_l\ self)\ (v_{m_l}\ self)\ y & \end{array}\right\} \qquad \text{(meth add 2)}$$
$$\mathsf{in\ obj}\langle \mathit{fix}(\mathrm{incgen}), \mathrm{incgen}, r - r.m_l, New\cup\{l\}, Redef-\{l\}, Expect\rangle$$

$$\mathsf{obj}\langle\{\ldots\},g,r,New,Redef,Expect\rangle \longleftarrow\!+\ \mathsf{obj}\langle\{m_i = v_{m_i}\}^{i\in I}, g'\rangle \rightarrow$$
$$\mathsf{let\ incgen} = \lambda self.$$
$$\left\{\begin{array}{ll} m_j = \lambda y.\ (g\ self).m_j\ y & {\scriptstyle j\in New} \\ m_k = \lambda y.\ (r.m_k\ self)\ (g'\ self).m_k\ y & {\scriptstyle k\in Redef} \\ m_i = \lambda y.\ (g'\ self).m_i\ y & {\scriptstyle i\in I-Redef} \end{array}\right\} \qquad \text{(obj comp)}$$
$$\mathsf{in\ obj}\langle \mathit{fix}(\mathrm{incgen}), \mathrm{incgen}\rangle$$

$$\mathsf{obj}\langle\{m_i = v_{m_i}\}^{i\in I}, g, \{\}, I, \emptyset, \emptyset\rangle \rightarrow \mathsf{obj}\langle\{m_i = v_{m_i}\}^{i\in I}, g\rangle \qquad \text{(completed)}$$

Fig. 7. Reduction rules for object completions

(mixin) rule turns a mixin *expression* into a mixin *value* (notice that all the other mixin operations, i.e., mixin application and mixin instantiation, are performed on mixin values). Given the parameter x for the constructor c of the mixin expression (we remind that c is a function that calculates the initializing values for the private field of the mixin, and for the generator of the future superclass), the mixin generator returns a record containing the following:

- a (partial) object generator gen which binds the private field of the methods m_j (newly defined by the mixin) to fieldinit (returned by the constructor). Recall that method bodies take parameters *field*, *self*, and, if it is a redefinition, also *next*. The output of gen has "dummy" method bodies in place of redefined and expected methods to enable the correct instantiation of incomplete objects. Intuitively, *self* must refer to all the methods: not only the new ones, but also the ones that are still to be added;
- the argument superinit for the superclass constructor, as returned by the mixin constructor c (the constructor subexpression c is a function of one argument which returns a record of two components: one is the initialization expression for the field fieldinit the other is the superclass generator's argument superinit);
- the redef component which contains a record of redefined methods that have their private field already bound to fieldinit (returned by the constructor), and are ready to have their *next* parameter bound to a method added to the object at run time, with their *self* still unbound. This record will be used during method addition and object composition to recover the actual body of the redefined methods, complete it, and insert it in the working part of the object.

The above generator is called "partial" since it returns an object that contains redefined and expected methods that cannot be invoked (present as "dummy" methods). The actual implementation of those methods can be provided by (meth add 1), (meth add 2), and/or (obj comp).

(new class) rule enables the creation of new objects from class definitions. It builds a function which, once passed an argument v, produces the *complete object* $\mathsf{obj}\langle \mathit{fix}(g\ v), (g\ v)\rangle$. $(g\ v)$ is the object generator, obtained by applying the class generator g to an argument v. This creates a function from *self* to a record of methods. $\mathit{fix}(g\ v)$ is the record of methods that can be invoked on that object, obtained by applying the fixed-point operator *fix* (following [20]) to $(g\ v)$ to bind *self* in method bodies and create a recursive record.

(new mixin) rule creates *incomplete objects* from mixin values. First, it applies the mixin generator Gen_m to an argument v, thus binding the private field of new and redefined methods and providing access to Gen_m's

gen and redef components. The mixin object generator g.gen is a function from *self* to a record of mixin methods, while g.redef is the record of the redefined mixin methods that have their fieldinit bound (*self* and *next* have still to be bound). The g.redef record is used for "remembering" the partial redefined method bodies for the future use. The application of the fixpoint operator to g.gen creates a recursive record of methods[4].

(obj sel) enables method invocation on a complete object.

(incobj sel) selects the method from the incomplete object. Note that each method m_i can be called provided that it only uses methods whose names are in *New* (and recursively these methods only call methods that are in *New*). This condition is not checked by the semantics since it is checked by the type system.

(mix app) rule evaluates the application of a mixin value to a class value and represents inheritance in our calculus. A mixin value is applied to a superclass value $\mathsf{classval}\langle g, \mathcal{M}\rangle$, where $\mathcal{M}$ 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, and $New \cup \mathcal{M}$ lists all the method names of the subclass. Using a class generator delays full inheritance resolution until object instantiation time when *self* becomes available.

The class generator 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 the (new class) rule). *Gen* first calls $Gen_m(x)$ to compute *mixinrec*, which is used first to compute the mixin object generator *mixingen*, a function from *self* to a record of mixin methods. Next, it is used to compute *mixinred*, which provides the record of redefining methods from the mixin. Then, *Gen* calls the superclass generator g, passing argument *mixinrec*.superinit, to obtain a function *supergen* from *self* to a record of superclass methods. Finally, *Gen* builds a function from *self* that returns a record containing *all* 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 (as checked by the type system). 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 $(supergen\ self).m_k$ by *Gen*, which is then passed to $(mixinred.m_k)\ self$. Notice that at this stage, all methods have already received a binding for the private field. Moreover, for all methods in all generator functions, the method bodies are wrapped inside $\lambda y.\cdots y$ to delay evaluation in our call-by-value calculus.

The next four rules in Figure 7 are the basic rules for manipulating the incomplete objects, i.e., they enable completing them with the method definitions that they need either as expected or redefined.

(meth add 1) rule adds to an incomplete object a method m_l that some other methods expect. The function incgen maps *self* to a record of methods, where new method definitions are taken from the object generator g, the redefining and expected (excluding m_l) methods remain "dummy" and the method m_l is added. Therefore, applying the *fix* operator to incgen produces a recursive record of methods with bound *self* and implicitly enables invocation of the methods that could have not been invoked before. The *incgen* function is part of the reduct because it must be carried along in the evaluation process, in order to enable future method additions and/or object compositions.

(meth add 2) rule is similar to the previous one, the difference being that now a method is added in order to complete a redefining method m_l, acting as its *next*. Therefore the definition of the redefined method is not "dummy" anymore, but gets a new body $m_l = \lambda y.\ (r.m_l\ self)\ (v_{m_l}\ self)\ y$. The body of m_l is taken from r (it is already bound to fieldinit) and $(v_{m_l}\ self)$ is passed to it as *next*. Naturally, this method becomes fully functional, therefore its definition is removed from r, and the index l is removed from *Redef* and added to *New*.

The only requirement for m_l both in rules (meth add 1) and (meth add 2) is that the body v_{m_l} must be a function of *self*.

(obj comp) rule combines two objects in such a way that the new added object o_2 (which must be, in turn, already complete) completes the incomplete object o_1 and makes it fully functional. The record of methods in *incgen* is built by taking the new methods from the incomplete object o_1 (these are the only methods

[4] Those methods that do not invoke any expected method and/or that have their reference to their *next* already resolved can be called on this recursive record component of the newly produced incomplete object.

that are fully functional in this object), binding the *next* parameter in redefining methods from o_1, and taking the expected methods from the complete object o_2. During object composition, the object o_2 used for the completion gets *self* rebound to the new resulting object (this is the reason why we need to keep the generator also for complete object values). This rebinding automatically enables dynamic binding of methods that are redefined even when called from within the methods of o_2. After completion, it is possible to invoke all the methods on the new created object.

(completed) rule transforms an incomplete object, for which all the missing methods are provided, into a corresponding complete one.

It might be tempting to argue that object composition is just syntactic sugar, i.e., it can be derived via an appropriate sequence of method additions, but this is not true. In fact, when adding a method, the method does not have a state, while a complete object used in an object composition has its own internal state (i.e., it has a private field, properly initialized when the complete object was created via a "new" from a class, or when part of it was created via a "new" from a mixin) . Being able to choose to complete an object via the composition or via a sequence of method additions (of the same methods appearing in the (complete) object used in the composition) gives our calculus an extra bit of flexibility.

5 Conclusions

In this work we integrate some features from class-based and object-based programming realities. As a step forward towards designing a flexible object-oriented language, we decided that the promising starting point could be the combination of mixin-based inheritance with constructs for manipulating incomplete objects, instantiated from mixins seen as "incomplete" classes. Incomplete objects can be completed via method addition and object composition, which either add one missing method to the incomplete object, or compose it with another (complete) object.

We plan to add higher-order mixins to the calculus, along the line of [6]. Moreover, we want to study a form of object-based method *override* that would work well in our hybrid setting, and a more general form of method addition. Finally, incomplete objects 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.

Generally speaking, all object-based calculi can be seen as calculi of incomplete objects, especially when in presence of a method addition operation (one example is [26]). However, an explicit form of incomplete objects was introduced in [9], where an extension of [26] is presented. In this work, "labelled" types are used to collect information on the mutual dependencies among methods (labels list the names of the methods that a given method might invoke), in order to have a safe subtyping in width. Labels are also used to implement the notion of *completion* which enables adding methods in an arbitrary order allowing the typing of methods that refer to methods not yet present in the object, thus supporting a form of incomplete objects.

However, to the best of our knowledge, there exist no attempts other than ours to instantiate mixins in order to obtain prototypical incomplete objects within a hybrid class-based/object-based framework, even though a similar approach might be explored by G. Boudol [14] in the setting presented in [13].

Acknowledgments. The authors would like to thank the anonymous referees for some useful suggestions on how to improve the presentation of the calculus.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of java with mixins. In *Proc. ECOOP 2000*, pages 154–178. LNCS 1850, Springer-Verlag, 2000.
3. D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of Java classes. In *PPDP'99 - Principles and Practice of Declarative Programming*, pages 189–200. ACM, 2002.
4. C. Anderson, F. Barbanera, M. Dezani-Ciancaglini, and S. Drossopoulou. Can Addresses be Types? (a case study: objects with delegation). In *WOOD'03*, ENTCS. Elsevier, 2003. To appear.
5. C. Anderson and S. Drossopoulou. δ - an imperative object based calculus. Presented at the workshop USE in 2002, Malaga, `http://www.binarylord.com/work/delta.pdf`, 2002.
6. L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Higher-Order Mixins and Classes. In *SAC, Special Track on Programming Languages*. ACM Press, 2004. Poster paper, to appear.
7. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Mobile Code. In F. Arbarb and C. Talcott, editors, *Proc. of Coordination Models and Languages*, number 2315 in LNCS, pages 56–71. Springer, 2002.

8. L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Classes and Mixins. In *Proc. FOOL '03*, 2003.
9. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. A Subtyping for extensible, incomplete objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.
10. V. Bono, F. Damiani, and P. Giannini. A calculus for "environment-aware" computation. In *F-WAN'02*, volume 66.3 of *ENTCS*. Elsevier, 2002.
11. V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *Proc. ECOOP '98*, volume 1445 of *LNCS*, pages 462–497, 1998.
12. 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.
13. G. Boudol. The recursive record semantics of objects revised. In *Proc. ESOP '01*, pages 269–283. LNCS 2028, Springer-Verlag, 2001.
14. G. Boudol. Private communication, 2002.
15. 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.
16. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
17. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.
18. K. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.
19. K. B. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.
20. W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
21. E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. POPL '91*, pages 233–244, 1991.
22. F. Damiani and P. Giannini. Alias types for "environment-aware" computation. In *WOOD'03*, ENTCS. Elsevier, 2003. To appear.
23. 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.
24. R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ICFP '98*, pages 94–104, 1998.
25. K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996.
26. K. Fisher, F. Honsell, and J. C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994. Preliminary version appeared in Proc. *LICS '93*, pp. 26–38.
27. K. Fisher and J. Reppy. Inheritance-based subtyping. *Information and Computation*, 177(1):28–55, 2002.
28. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
29. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
30. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *Proc. ESOP '00*, volume 2305 of *LNCS*, pages 6–20. Springer-Verlag, 2002.
31. 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):1–30, 1996.
32. 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.
33. A. Patel. *Obstacl: a language with objects, subtyping, and classes*. PhD thesis, Stanford University, 2001.
34. J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 172(1):2–28, 2002. A preliminary version appeared in FOOL5.
35. J. Vouillon. Combining subsumption and binary methods: An object calculus with views. In *Proc. POPL '01*, pages 290–303, 2001.
36. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.