Safe and Flexible Objects *

Lorenzo Bettini[1] Viviana Bono[2] Silvia Likavec[2]
[1]Dipartimento di Sistemi ed Informatica, Università di Firenze, bettini@dsi.unifi.it
[2]Dipartimento di Informatica, Università di Torino,{bono,likavec}@di.unito.it

ABSTRACT

We design a calculus where objects are created by instantiating classes, as well as mixins. Mixin-instantiated objects are "incomplete objects", that can be completed in object-based fashion. The combination of class-based features with object-based ones offers some flexible programming solutions. The fact that all objects are created from fully-typed constructs is a guarantee of controlled (therefore reasonably safe) behavior.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Theory

Keywords: Mixins, Incomplete Objects, Language Design, Types

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. For insightful discussions about the differences between the two paradigms we refer the reader to the books of Abadi-Cardelli [1] and of Bruce [10].

Our aim is to design a calculus that combines class-based features with object-based ones, trying to fit into one setting the "best of both worlds", discipline and flexibility first of all. Object-based objects can 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 is 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 may be completed via *method addition* and/or *object composition*.

In other words, it is possible to design class hierarchies via mixin application, but also to experiment with prototypical incomplete objects. Section 3 presents some scenarios where it seems useful to add new features to existing objects by means of functions, without having to write new mixins for this purpose only. ("Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition" [15].)

The main contribution of the present paper is a type system for the core calculus of mixins and incomplete objects [4], which ensures safe behavior of objects, both complete and with some missing components. The combination of class-based and object-based features proved to be a good programming environment. An extended abstract (without the presentation of the type system and the related properties) is included in the informal proceedings of FOOL 11 [4].

2. The Mixin Calculus

Mixins [9, 14] are (sub)class definitions parameterized over a superclass and were introduced as an alternative to some forms of multiple inheritance. 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. In this paper the term *mixin* refers to *mixin classes*, as opposed to *mixin modules* (modules supporting deferred components [2, 16]).

We extend the core calculus of classes and mixins of [8] 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 [12, 11]) or via *object composition*.

Our formal design choices are strongly based on the ideas presented in [8, 18]. 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 mixin constraint includes information on which methods the class must contain, whereas negative constraint, i.e., which methods the class must not contain, is checked by the type system at mixin application time. In this 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. The paper deals with incomplete objects remaining faithful to the design principles of [8]. Meaningful differences and extensions to the syntax of the original calculus are commented on below.

Starting from the imperative calculus of records, functions,

*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. The funding bodies are not responsible for any use that might be made of the results presented.

SACAE05, March 13-17, 2005, Santa Fe, New Mexico, USA.

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

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

classes, and mixins of [8], we add the constructs to work with incomplete objects.

The lambda-calculus related forms in Figure 1 are standard. For the explanation of the expressions dealing with side-effects, the heap-related expressions, and the fixpoint operator *fix*, we refer the reader to [8, 21]. We describe below the forms related to mixins and incomplete objects.

- $\mathsf{classval}\langle v_g, \mathcal{B}\rangle$ is a *class value*, the result of mixin application. The function v_g is the *generator* used to generate its instances, and the set $\mathcal{B}$ contains the indices of all the methods defined in the class.
- a *mixin* contains three sorts of method declarations: *new* methods (m_j), which are the newly introduced methods by the mixin seen as a subclass, *redefining* methods (m_k), which wait for a superclass containing methods with the same name to be redefined, and provide the overriding body, and *expected* method names (m_i), which are methods not implemented by the mixin but must be provided by the superclass (since they might be used by new and redefining methods). 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. New and redefined methods are distinguished in the mixin implementation, since 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). Each method body $v_{m_{j,k}}$ 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 (old) redefined method from the superclass. For the sake of simplicity, we consider only one field for each mixin, but this is not a restriction, as the field could be a tuple. Also, the calculus does not enforce the field to be available to all the methods of the mixin, but this is easily obtained by using an imperative variable for the field. If so, the field behaves like a proper private instance variable (being non-accessible, not only non-visible). The constructor value v_c is a function of one argument that returns a record of two components: the fieldinit value used to initialize the private field, and the superinit value passed as an argument to the superclass constructor. When evaluating a mixin, v_c is used to build the generator.
- $\mathsf{mixinval}\langle v_g, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle$ is a *mixin value*, the result of mixin evaluation. The generator v_g for the mixin is a "partial generator", of incomplete objects, used also in the $\diamond$ operation evaluation, where it is appropriately composed with the class generator. The sets $\mathcal{N}$, $\mathcal{R}$ and $\mathcal{E}$ contain the indices of new, redefining, and expected methods defined in the mixin.
- new *e* creates a function that returns a new object (incomplete, in the mixin case).
- $e_1 \diamond e_2$ is the application of mixin value e_1 to class value e_2, producing a new class value that is a subclass of e_2.
- $e_1 \longleftarrow m_i = e_2$ is the method addition operation: it adds the definition of method m_i with body e_2 to the (incomplete) object to which e_1 evaluates. Method m_i must be declared as either expected or redefined in e_1.
- $e_1 \longleftarrow e_2$ is the object composition operation: it composes the (incomplete) object to which e_1 evaluates with the complete object to which e_2 evaluates.
- $\mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in I}, v_g\rangle$ 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, kept also for complete objects, since they can be used to complete the incomplete ones.
- $\mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in I}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle$ is an incomplete object. $\{m_i = v_{m_i}\}^{i\in I}$ is a record of methods, v_g is a generator function, r is a record containing redefining methods which will be used when a *next* for them becomes available during method addition or object composition. When the sets $\mathcal{R}$ and $\mathcal{E}$ become empty (and so does the record of redefining methods) the incomplete object becomes a complete object.

Mixins are first-class citizens in our calculus, i.e., all the usual operations are allowed on them. 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 are created by mixin application. Mixin values result from evaluation of mixins and are partially evaluated incomplete classes useful when creating incomplete objects. They have exactly the same types as the mixins they are evaluated from (see Section 4). Object forms can be created by class and mixin instantiation. Notice that objects are not simply records, they are tuples, in order to be able to deal with incompleteness. Still, methods are all contained in a record-shaped component of the tuple.

We define the root of the class hierarchy, class *Object*, as a predefined class value $Object \triangleq \mathsf{classval}\langle\ \lambda\_.\lambda\_.\{\},\ [\ ]\ \rangle$ necessary for a uniform treatment of all the other classes.

The formal operational semantics of our calculus is as in [4], therefore it extends the semantics of the core calculus of classes and mixins of [8]. It is a set of rewriting rules including standard rules for a call-by-value lambda calculus with stores (in our case *Reference ML* of Wright and Felleisen [21]), and 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-objects principle (see [8]). Our calculus uses an early-binding strategy where the fixpoint operator that binds self to the host object is applied either at object creation time, or at method addition/object composition time. A late-binding strategy (where the host object is substituted to self at method invocation time) is less efficient from an implementation point of view, and necessary only in the presence of polymorphic self-types (i.e., *MyType*, [12]). For the lack of space, we only give an informal description of the rules to deal with mixins and objects. The full set of rules and their detailed description can be found in [4].

Mixin *expressions* evaluate into mixin *values*, since mixin application and mixin instantiation, are performed on mixin values. In this process, a *mixin generator* is produced, which is a function that returns a record of "mixin pre-methods", where the private field is already bound to its initializing value, but *self* is still unbound (it will be bound at object-creation time).

If we apply a mixin (value) to a class (value) via the *mixin application* operation, we obtain a new fully-fledged subclass. All methods of the superclass that are not redefined by the mixin are *inherited* by the subclass. These methods include all the methods that are expected by the mixin (as checked by the type system). Methods defined by the mixin are taken intact from the mixin. The mixin generator function hinted above plays a crucial role in the construction of the class generator representing the subclass (value).

By instantiating a class, we obtain a complete object where all the methods are fully functional. In particular, a complete object is essentially a pair, where the first component contains the methods in their invocable form, and the second component represents the generator (which is a sort of set of "pre-methods") that may be used if the complete object is composed with an incomplete one. When

we instantiate a mixin we obtain an incomplete object that can be "incomplete" in two respects: (*i*) it may need some expected methods; (*ii*) it may contain redefining methods that need the methods with the functionality of their *next*.

Method invocation might be also allowed on incomplete objects, but only on those methods that are already "complete", i.e., the ones that do not need a *next* and do not use either expected or other incomplete methods. It would be necessary to implement a sort of "transitive closure", based on a global analysis technique, to list, for each method, the dependencies from other methods, but since this feature is essentially an implementation detail, we leave it out from this version of the calculus.

Completion can happen in two ways: (*i*) via *method addition*, that can add one of the *expected* methods or one of the missing *nexts* (methods acting as the superclass' implementation of the methods with the same names); (*ii*) via *object composition*, that takes an incomplete object and composes it with a complete one that contains all the required methods. The type system (Section 4) ensures that all the method additions and object compositions are type safe. 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.

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 the 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.

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 "new" from a class, or when part of it was created via "new" from a mixin). Being able to choose to complete an object via 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.

3. Examples

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

For readability, we will use here a slightly simplified syntax with respect to the calculus presented in Section 2: (*i*) the methods' parameters are listed 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*) references are not made 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. Finally, $x \longleftarrow + \ e$ should be intended, informally, as $x := (x \longleftarrow + \ e)$.

3.1 Object Completion via Method Addition

In the first example, we present a scenario where it is useful to add some functionalities to existing objects without writing new mixins and creating related classes only for this purpose. Let us consider the development of an application that uses 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* [15] is useful for implementing these scenarios, since it allows parameterization of 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 convenient 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 further functionalities needed by the prototype). For instance, we could implement the solution as in

```
let Button =                    let MenuItem =                  let ShortCut =
 mixin                           mixin                           mixin
  method display = ...            method show = ...               method setEnabled = ...
  method setEnabled = ...         method setEnabled = ...         expect onClick;
  expect onClick;                 expect onClick;                 ...
  ...                             ...                             end in
 end in                          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 ←+ (OnClick = ClickHandler);
          button.display();
          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);
```

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 is then 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. At this point the object is completed and we can call methods on it. 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 from 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 ←+ (OnClick = ClickHandler);
   funnybutton.display();
   funnybutton.setEnabled(true); // now it is safe to use it
   toolbar.addButton(funnybutton);
```

In fact, the mixin `FunnyButton` does not simply expect the method `onClick`, it expects to redefine this 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 an 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* [15]. 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. 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 the following, we give the definitions of `Compress` mixin and `Buffer` mixin that 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*.

```
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 ...
```

Streams are created by composing streams with advanced functionalities, that are incomplete objects (i.e., instances of the appropriate mixin), like `Compress`, with streams with basic functionalities like, `File`. The power of object composition can be seen when we compose more than one stream in a chain of objects. For example, the following instructions create a UUEncoded, compressed file stream where I/O is buffered:

```
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, 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 to, e.g., user's requests: it can return a simple `Compress` object, or a `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
   logger.doLog("logging started...");
   logger.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 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
      logger.doLog("logging started...");
      logger.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 (see Section 4) will ensure that all object compositions are type safe.

4. Type System

We extend the work of [4] with a type system. Besides functional, record, and reference types, our type system has class types, mixin types, and object types (both for complete and incomplete objects):

$$\begin{array}{rcl}\tau ::= & \iota \mid \tau_1 \to \tau_2 \mid \tau\ \mathsf{ref} \mid \{m_i : \tau_{m_i}\}^{i\in I} \mid \mathsf{class}\langle \tau, \Sigma_{\mathcal{B}}\rangle \\ \mid & \mathsf{mixin}\langle \tau_1, \tau_2, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle \mid \mathsf{obj}\langle\Sigma\rangle \mid \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle\end{array}$$

where ι is a constant type, $\to$ is the functional type operator, τ ref is the type of locations containing a value of type τ. Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i}\}^{i\in I}$, $I \subseteq \mathbb{N}$. We use the following notation:

- if $m_i : \tau_{m_i} \in \Sigma$ we say that the *label* m_i *occurs* in Σ (with type τ_{m_i}). $\mathcal{L}(\Sigma)$ denotes the set of all the labels occurring in Σ;
- $\Sigma_1/\Sigma_2 = \{m_i : \tau_{m_i} \mid m_i : \tau_{m_i} \in \Sigma_1 \ \wedge\ m_i \in \mathcal{L}(\Sigma_2)\}$ is the operator that refers to the method names appearing in both records, giving them the types present in the first one.

$$\frac{\Gamma \vdash v_g : \gamma \to \{m_i : \tau_{m_i}\}^{i\in\mathcal{B}} \to \{m_i : \tau_{m_i}\}^{i\in\mathcal{B}}}{\Gamma \vdash \mathsf{classval}\langle v_g, \mathcal{B}\rangle : \mathsf{class}\langle \gamma, \{m_i : \tau_{m_i}\}^{i\in\mathcal{B}}\rangle} \quad (\mathsf{T\ class\ val})$$

$$\frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \{m_i : \tau_{m_i}\}\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \to \mathsf{obj}\langle \{m_i : \tau_{m_i}\}\rangle} \quad (\mathsf{T\ class\ inst})$$

$$\frac{\Gamma \vdash \{m_i = v_{m_i}\}^{i\in I} : \{m_i : \tau_{m_i}\}^{i\in I} \quad \Gamma \vdash v_g : \{m_i : \tau_{m_i}\}^{i\in I} \to \{m_i : \tau_{m_i}\}^{i\in I}}{\Gamma \vdash \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in I}, v_g\rangle : \mathsf{obj}\langle \{m_i : \tau_{m_i}\}^{i\in I}\rangle} \quad (\mathsf{T\ obj})$$

$$\frac{\Gamma \vdash e : \mathsf{obj}\langle \Sigma\rangle \qquad m_i : \tau_{m_i} \in \Sigma}{\Gamma \vdash e.m_i : \tau_{m_i}} \quad (\mathsf{T\ sel})$$

Figure 2: Typing rules for class related forms

Typing environments are defined as: $\Gamma ::= \varepsilon \mid \Gamma, x:\tau \mid \Gamma, \iota_1 <: \iota_2$ where $x \in Var$, τ is a well-formed type, ι_1, ι_2 are constant types, and $x, \iota_1 \notin dom(\Gamma)$.

Typing judgments are the following:

$\Gamma \vdash \tau_1 <: \tau_2$ τ_1 is a subtype of τ_2 $\quad \Gamma \vdash e : \tau$ e has type τ.

Typing rules for class and complete object related forms are given in Figure 2. In rule ($\mathsf{T\ class\ val}$), $\mathsf{class}\langle \gamma, \Sigma_{\mathcal{B}}\rangle$ is the class type where γ is the type of the generator's argument and $\Sigma_{\mathcal{B}} = \{m_i : \tau_{m_i}\}$ is a record type representing the interface of the objects instantiable from the class. The type of a complete object is the record of its method types (rule ($\mathsf{T\ obj}$)). Notice that objects instantiated from class values do not have a simple record type Σ, but an object type $\mathsf{obj}\langle\Sigma\rangle$. This is useful for distinguishing standard complete objects, which can be used for completing incomplete objects, from their internal auto-reference *self*, that has type Σ. Note also that in the object expression, the second component v_g is a function from *self* to *self* because it works on the first component of the object, which is the record of object's methods. The only operation allowed on complete objects is method selection and it is typed as ordinary record component selection (rule ($\mathsf{T\ sel}$)).

Typing rules for lambda expressions are standard. Typing rules for expressions dealing with imperative side-effects via stores and the rules for typing classes and records can be found in [8]. We present in Figure 3 only the typing rules for mixin-related forms. In the mixin type $\mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle$:

- γ_b is the expected argument type of the superclass generator,
- γ_d is the exact argument type of the mixin generator,
- $\Sigma_{\mathcal{N}} = \{m_j : \tau^{\downarrow}_{m_j}\}$ are the exact types of the new methods introduced by the mixin,
- $\Sigma_{\mathcal{R}} = \{m_k : \tau^{\downarrow}_{m_k}\}$ are the exact types of the methods redefined by the mixin,
- $\Sigma_{\mathcal{E}} = \{m_i : \tau^{\uparrow}_{m_i}\}$ are the types of the methods expected to be supported by the superclass to which the mixin is applied,
- $\Sigma_O = \{m_k : \tau^{\uparrow}_{m_k}\}$ are the types assumed for the old bodies (which will be provided by a superclass) of the methods redefined by the mixin.

The rules ($\mathsf{T\ mixin}$) and ($\mathsf{T\ mixin\ val}$) assign the same type to their respective expressions. The type assigned to an incomplete object is similar to the type of the mixin the object is the instance of, but it does not contain information about the constructor (see rule ($\mathsf{T\ inc\ obj}$)), since the constructor has already been called when the incomplete object has been created. In object values, the type of the private field disappears since it has already been bound during mixin instantiation (while the type of *self* is still present since it will be bound later during method addition or object composition). Notice that in the rule ($\mathsf{T\ inc\ obj}$) the record of methods includes also expected methods ($i \in \mathcal{N} \cup \mathcal{R} \cup \mathcal{E}$). This may seem to contradict the "nature" of expected methods. Indeed, such methods in an incomplete object are "dummy" in the sense that they are of the shape $m = \lambda self.self.m$ (see [4] for the detailed explanation of the operational semantics), and they will never be called,

($\mathsf{T\ mixin}$)
$j \in \mathcal{N}$: $\Gamma \vdash v_{m_j} : \eta \to \Sigma \to \tau^{\downarrow}_{m_j}$ $\quad k \in \mathcal{R}$: $\Gamma \vdash v_{m_k} : \eta \to \Sigma \to \tau^{\uparrow}_{m_k} \to \tau^{\downarrow}_{m_k}$
$\Gamma \vdash v_c : \gamma_d \to \{\mathsf{fieldinit} : \eta, \mathsf{superinit} : \gamma_b\}$

$$\Gamma \vdash \begin{array}{l}\mathsf{mixin}\\ \mathsf{method}\ m_j = v_{m_j};\ {}^{(j\in\mathcal{N})}\\ \mathsf{redefine}\ m_k = v_{m_k};\ {}^{(k\in\mathcal{R})}\\ \mathsf{expect}\ m_i;\ {}^{(i\in\mathcal{E})}\\ \mathsf{constructor}\ v_c;\\ \mathsf{end}\end{array} : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle$$

($\mathsf{T\ mixin\ val}$)
$$\frac{\Gamma \vdash v_g : \gamma_d \to \{\mathsf{gen} : \Sigma \to \Sigma, \mathsf{superinit} : \gamma_b, \mathsf{redef} : \{m_k : \Sigma \to \tau^{\uparrow}_{m_k} \to \tau^{\downarrow}_{m_k}\}^{k\in\mathcal{R}}\}}{\Gamma \vdash \mathsf{mixinval}\langle v_g, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle}$$

($\mathsf{T\ mixin\ inst}$)
$$\frac{\Gamma \vdash e : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma_d \to \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle}$$

($\mathsf{T\ inc\ obj}$)
$$\frac{j \in \mathcal{N}\text{: } \Gamma \vdash v_{m_j} : \tau^{\downarrow}_{m_j} \quad k \in \mathcal{R}\text{: } \Gamma \vdash v_{m_k} : \tau^{\downarrow}_{m_k} \quad i \in \mathcal{E}\text{: } \Gamma \vdash v_{m_i} : \tau^{\uparrow}_{m_i} \quad \Gamma \vdash v_g : \Sigma \to \Sigma \quad \Gamma \vdash r : \{m_k : \Sigma \to \tau^{\uparrow}_{m_k} \to \tau^{\downarrow}_{m_k}\}^{k\in\mathcal{R}}}{\Gamma \vdash \mathsf{obj}\langle \{m_l = v_{m_l}\}^{l\in\mathcal{N}\cup\mathcal{R}\cup\mathcal{E}}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle : \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle}$$

($\mathsf{T\ mix\ app}$)
$$\frac{\Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_O\rangle \quad \Gamma \vdash e_2 : \mathsf{class}\langle \gamma_c, \Sigma_{\mathcal{B}}\rangle \quad \Gamma \vdash \gamma_b <: \gamma_c \quad \Gamma \vdash \Sigma_{\mathcal{B}} <: (\Sigma_{\mathcal{E}} \cup \Sigma_O) \quad \Gamma \vdash \Sigma_{\mathcal{R}} <: \Sigma_{\mathcal{B}}/\Sigma_{\mathcal{R}} \quad \mathcal{L}(\Sigma_{\mathcal{B}}) \cap \mathcal{L}(\Sigma_{\mathcal{N}}) = \emptyset}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{class}\langle \gamma_d, \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup (\Sigma_{\mathcal{B}} - (\Sigma_{\mathcal{B}}/\Sigma_{\mathcal{R}}))\rangle}$$

where in all the rules
$\Sigma = \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}}$, $\Sigma_{\mathcal{N}} = \{m_j : \tau^{\downarrow}_{m_j}\}$, $\Sigma_{\mathcal{R}} = \{m_k : \tau^{\downarrow}_{m_k}\}$, $\Sigma_{\mathcal{E}} = \{m_i : \tau^{\uparrow}_{m_i}\}$, $\Sigma_O = \{m_k : \tau^{\downarrow}_{m_k}\}$, $\tau^{\uparrow}_{m_i}$ and $\tau^{\uparrow}_{m_k}$ are inferred from method bodies

Figure 3: Typing rules for mixin related forms

since the typing prohibits to invoke methods on incomplete objects. "Dummy" methods are a technical trick that enables 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). When typing an incomplete object value and a mixin value, "dummy" methods allow us to assign the type $\Sigma \to \Sigma$ to the generator v_g (the generator being a function from *self* to *self*). In fact, the body of "dummy" methods simply calls the homonym method on *self*, so the type inferred for expected and redefined methods will be consistent with the types of "dummy" method bodies (and so with the types of expected and *next* methods sought by their sibling methods). If the "dummy" method "trick" was not used, the *fix* operator could not be applied to generate an incomplete object.

In the rule ($\mathsf{T\ mix\ app}$), $\Sigma_{\mathcal{B}}$ contains the type signatures of all the methods supported by the superclass to which the mixin is applied and $\Sigma_{\mathcal{B}}/\Sigma_{\mathcal{R}}$ are the superclass methods redefined by the mixin (superclass may have more methods than required by the mixin constraints). Actual types of the superclass methods must be subtypes of those expected by the mixin, while the types of actual implementations of the methods in the superclass must be supertypes of the ones redefined by the mixin. The resulting class contains the signatures of all the methods defined by the mixin, and inherited from the superclass (with redefining methods from the mixin).

Figure 4 shows the typing rules related to incomplete objects. A method m_l can be added to an incomplete object (rule ($\mathsf{T\ meth\ add\ 1}$)), only if this method is expected by the incomplete object and if its type is a subtype of the expected one. The added method completes the functionalities of some already present methods, and may invoke some of them as well. Therefore, m_l's *self* type Σ_1 imposes some constraints on the type of the incomplete

(T meth add 1)
$\Gamma \vdash e : \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_{\mathcal{O}} \rangle \quad m_l : \tau^{\uparrow}_{m_l} \in \Sigma_{\mathcal{E}} \quad \Gamma \vdash v_{m_l} : \Sigma_1 \to \tau_{m_l}$
$\Gamma \vdash \tau_{m_l} <: \tau^{\uparrow}_{m_l} \quad \Gamma \vdash (\Sigma_{\mathcal{N}} \cup \{m_l : \tau_{m_l}\} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}} - \{m_l : \tau^{\uparrow}_{m_l}\}) <: \Sigma_1$

$\Gamma \vdash e \longleftarrow\!+\ (m_l = v_{m_l}) : \mathsf{obj}\langle \Sigma', \Sigma_{\mathcal{N}} \cup \{m_l : \tau_{m_l}\}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} - \{m_l : \tau^{\uparrow}_{m_l}\} \rangle \Sigma_{\mathcal{O}}$

(T meth add 2)
$\Gamma \vdash e : \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}, \Sigma_{\mathcal{O}} \rangle \quad m_l : \tau^{\uparrow}_{m_l} \in \Sigma_{\mathcal{O}} \wedge m_l : \tau^{\downarrow}_{m_l} \in \Sigma_{\mathcal{R}}$
$\Gamma \vdash v_{m_l} : \Sigma_1 \to \tau_{m_l} \quad \Gamma \vdash \tau^{\downarrow}_{m_l} <: \tau_{m_l} <: \tau^{\uparrow}_{m_l} \quad \Gamma \vdash (\Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}}) <: \Sigma_1$

$\Gamma \vdash e \longleftarrow\!+\ (m_l = v_{m_l}):$
$\mathsf{obj}\langle \Sigma', \Sigma_{\mathcal{N}} \cup \{m_l : \tau_{m_l}\}, \Sigma_{\mathcal{R}} - \{m_l : \tau^{\downarrow}_{m_l}\}, \Sigma_{\mathcal{E}} \rangle \Sigma_{\mathcal{O}} - \{m_l : \tau^{\uparrow}_{m_l}\}$

(T obj comp)
$\Gamma \vdash e_1 : \mathsf{obj}\langle \Sigma_1, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle \Sigma_{\mathcal{O}} \quad \Gamma \vdash e_2 : \mathsf{obj}\langle \Sigma_2 \rangle$
$\Gamma \vdash \Sigma_2 <: (\Sigma_{\mathcal{E}} \cup \Sigma_{\mathcal{O}}) \quad \Gamma \vdash \Sigma_{\mathcal{R}} <: (\Sigma_2 / \Sigma_{\mathcal{R}}) \quad \mathcal{L}(\Sigma_2) \cap \mathcal{L}(\Sigma_{\mathcal{N}}) = \emptyset$

$\Gamma \vdash e_1 \longleftarrow\!+\ e_2 : \mathsf{obj}\langle \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup (\Sigma_2 - (\Sigma_2 / \Sigma_{\mathcal{R}})) \rangle$

(T compl)
$\Gamma \vdash \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i \in I}, v_g, \{\}, I, \emptyset, \emptyset \rangle : \mathsf{obj}\langle \Sigma, \emptyset, \emptyset, \emptyset \rangle$

$\Gamma \vdash \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i \in I}, v_g \rangle : \mathsf{obj}\langle \Sigma \rangle$

Figure 4: Typing rules for incomplete object-related forms

object that m_l is supposed to complete. Hence, the incomplete object must provide all the methods listed in Σ_1, on which the added method is parameterized. Σ_1 is inferred from m_l's body. The rule for adding a *next* method to complete a method m_l that is redefined in the incomplete object is similar and we omit its explanation due to the lack of space.

There are two important issues to notice. The first issue is that we do not need any form of recursive types because we do not use a polymorphic *MyType* to type *self* (see, for instance, [12]). This prevents the type system from typing binary methods, but it still allows it to type methods that modify *self*, which can be modelled as "void" methods that return nothing. The second issue concerns the subtype relation: we do not have subtyping for object types $\mathsf{obj}\langle \ldots \rangle$ (even though there is a subtyping rule for record types). This is in order to avoid conflicts between the form of object-based inheritance, that arises by object composition, and subtyping. In fact, subtyping-in-width may introduce unwanted name-clashes that can be dealt with using an approach similar to the *privacy-via-subsumption* approach given in [19]. Subtyping for complete and incomplete objects is work-in-progress and will appear elsewhere.

5. Conclusions

In this work we integrate some aspects from the mixin-based and the object-based programming paradigms: mixins are seen as "incomplete classes", and their instances as "incomplete objects" to be completed in an object-based fashion. We would like to point out that our purpose was not to model directly any form of delegation-based languages, but merely to explore the point of view of mixins seen as incomplete classes. Future work may include some thoughts on whether mixin-based incomplete objects can be useful in a delegation-based realm.

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. For example, a program "reaches a stuck state" if a method call is attempted on an expression that does not evaluate to an object. We prove that if the evaluation for a 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 typable, and, via a subject reduction property, we establish that if a program is typable, then it evaluates to a value, under the condition that the program does not diverge. The metatheory for the present system, and in particular the subject reduction property, are extensions of the ones in [6] (Chapter 9). The formal definitions and properties were analyzed in detail, and can be found in [17].

An explicit form of incomplete objects was introduced in [7], where an extension of Lambda Calculus of Objects of [12] is presented. In this work, "labelled" types are used to collect information on the mutual dependencies among methods, enabling a safe subtyping in width. Labels are also used to implement the notion of *completion* which permits 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.

Traits have been proposed in [20] as an alternative to class and mixin inheritance to enhance code reuse in object-oriented programs: they are collections of methods that can be used as "building blocks" for assembling classes. Traits are concerned with the reuse of behavior, while our main concern is the composition of objects together with their state. An interesting future direction is the study of incomplete objects using traits instead of mixins, starting from the typed calculus of traits [13].

We plan to add to our calculus the possibility to combine two mixins, thus introducing *higher-order mixins* (mixins that can also be applied to other mixins yielding other mixins), along the lines of [3]. This integration looks rather smooth, and it would result in a rather complete mixin-based setting. Moreover, we want to study a form of object-based method *override* and a more general form of method addition. Both these extensions will add issues to the subtyping problem, hinted at earlier in the paper (in Section 4). Finally, incomplete objects seem to be a natural feature to be added to MOMI [5], a coordination language where object-oriented mobile code is exchanged among the nodes of a network.

6. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
[2] 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.
[3] L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Higher-Order Mixins and Classes. In *Proc. of TYPES '03*, LNCS, 2003. To appear.
[4] L. Bettini, V. Bono, and S. Likavec. A core calculus of mixin-based incomplete objects. In *FOOL 11*, 2004.
[5] 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.
[6] V. Bono. *Type Systems for the Object Oriented Paradigm*. PhD thesis, Università di Torino, 1999.
[7] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. A Subtyping for extensible, incomplete objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.
[8] 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.
[9] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.
[10] K. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.
[11] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996.
[12] 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.
[13] K. Fisher and J. Reppy. A typed calculus of traits. In *FOOL 11*, 2004.
[14] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[16] 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.
[17] S. Likavec. *Types for object-oriented and functional programming languages*. PhD thesis, Università di Torino, 2005. Forthcoming.
[18] A. Patel. *Obstacl: a language with objects, subtyping, and classes*. PhD thesis, Stanford University, 2001.
[19] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 172(1):2–28, 2002. A preliminary version appeared in FOOL5.
[20] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *Proc. of ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.
[21] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.