POSTER ABSTRACT

A Core Calculus of Higher-Order Mixins and Classes *

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

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

ABSTRACT

We present an object-oriented calculus based on *higher-order* mixin construction via *mixin composition*, where some software engineering requirements are modelled in a formal setting, allowing to prove the absence of *message-not-understood* run-time errors.

1. INTRODUCTION

Recently, mixins are undergoing a renaissance (see, for example, [1, 2]), due to their nature of "incomplete" classes prone to be completed according to the programmer's needs. Mixins [4, 5] 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 or overriding certain sets of methods. The same mixin can be used to produce a variety of classes with the same functionality and behavior, since they all have the same sets of methods added and/or redefined. Also, the same mixin can be applied to the same class more than once, thus enabling incremental changes in the subclasses. The superclass definition is not needed at the time of writing the mixin definition. This minimizes the dependencies between superclass and its subclasses, as well as between class implementors and end-users, thus improving modularity. The uniform extension and modification of classes is instead absent from the classical class-based languages. Moreover, in class-based languages, parentage is determined statically at compile time instead of at run-time. Thus, using mixins avoids code duplication and improves modularity of the program.

In this work we extend the core calculus of classes and mixins of [3] with *higher-order* mixins. A mixin can: (*i*) be applied to a class to create a fully-fledged subclass; or (and this is the novelty with respect to [3]) (*ii*) be *composed* with another mixin in order to obtain yet another mixin with more functionalities. This enables cleaner modular and reusable object-oriented design via the construction of sophisticated class hierarchies, while keeping the good "software engineering" features of the original core calculus [3].

Our design decisions are strongly based on the choices that were made in [3]. Class hierarchies in a well-designed object-oriented program must not be fragile: if a superclass implementation changes but the specification remains intact, the implementors of the subclasses should not have to rewrite subclass implementations. This is only possible if object creation is modular. In particular, a subclass implementation should not be responsible for initializing inherited fields when a new object is created. Unlike many theoretical calculi for object-oriented languages, our calculus directly supports modular object construction. The mixin implementor only writes the local constructor for his own mixin. Mixin applications and compositions are reduced to generator functions that call all constructors in the inheritance chain in the correct order, producing a fully initialized object. To ensure that mixin inheritance can be statically type checked, the calculus employs constrained parameterization. From each mixin definition the type system infers a constraint specifying to which classes the mixin can be applied so that the resulting subclass is type-safe. The constraint includes both *positive* (which methods the class must contain) and *negative* (which methods the class must not contain) information. New and redefined methods are distinguished in the mixin implementation. From the implementor's viewpoint, a new method may have arbitrary behavior, while the behavior of a redefined method must be "compatible" with that of the old method it replaces. Having this distinction in the syntax helps mixin implementors avoid unintentional redefinitions of superclass methods and facilitates generation of the constraint for mixin's superclasses and for mixins that participate in mixin composition. The full paper and the metatheory for the calculus (that proves that "message-not-understood" errors do not occur in the calculus) can be found at `http://www.dsi.unifi.it/~bettini/mixin-high.html`.

2. OVERVIEW OF THE CALCULUS

Our calculus relies on the *Reference ML* [6], extended with objects (which are recursive records), and class- and mixin-related constructs. A class is a generator function, following Cook's seminal ideas. A mixin has the form:

$$
\begin{array}{l}
\mathsf{mixin} \\
\quad \mathsf{method}\ m_j = v_{m_j};\ {}^{(j \in New)} \\
\quad \mathsf{redefine}\ m_k = v_{m_k};\ {}^{(k \in Redef)} \\
\quad \mathsf{expect}\ m_i;\ {}^{(i \in Expect)} \\
\quad \mathsf{constructor}\ v_c; \\
\mathsf{end}
\end{array}
$$

Expressions $m_j = v_{m_j}$ are definitions of the new methods, $m_k = v_{m_k}$ are method redefinitions that will replace the methods with the same name in the superclass, and m_i are method (names) that the superclass is expected to implement. Each method body v_{m_j} (re-

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

SAC'04 March 14-17, 2004, Nicosia, Cyprus

spectively, v_{m_k}) is a function of the private *field* and of *self*, which will be bound to the newly created object at instantiation time. The v_{m_k}'s are also functions of *next*, which will be bound to the corresponding old method from the superclass. The v_c value in the constructor clause is a function that calculates two values, one to initialize the private field of an object at instantiation time, and one which will be passed as an argument to the superclass constructor. Constructors are used in: (*i*) mixin application, $e_1 \diamond e_2$, which denotes the application of mixin e_1 to class e_2, producing a subclass of e_2; (*ii*) mixin composition, $e_1 \bullet e_2$, that produces a new mixin taking components from both e_1 and e_2. The two mixins may partially complete each others' definitions, providing (some of) the missing components. Let *e* be the resulting mixin. The mixin e_2 acts as a "superclass" for e_1 (mirroring mixin application order), and, in particular, some of e_1 methods may override some of e_2 methods. Therefore, all of the new methods of the mixin e_1 are inserted in the resulting mixin *e* as new, while only the new methods of e_2 that are not redefined by e_1 become part of the new methods of *e* (name clashes are checked by the type system at compile time). As far as redefining methods are concerned, the methods specified as redefining in e_1 can override: some "new" methods of e_2 (these methods become part of the new methods of *e*), some "redefining" methods of e_2 (these methods become redefining methods of *e*, as the redefinition with the e_2 bodies is delayed, while e_1 has already performed a sort of "internal" redefinition over e_2), and (even if only virtually) some of the "expected" methods from e_2 (these methods become redefining methods of *e*). As in [3], we define the root of the class hierarchy, class *Object*, as a predefined class value.

3. EXAMPLE OF MIXIN INHERITANCE

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

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

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

let Random = mixin
  method lseek = ...
  expect write;
  expect read;
 end in

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

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

From the definition of `Encrypted`, the type system infers the types of the methods that the mixin wants to redefine: any class to which `Encrypted` is applied must contain *write* and *read* methods whose types must be supertypes of those given to *write* and *read*, in the definition of `Encrypted`. In `RandomFile` they are declared as *expected* and they are used within the method *lseek*. Once again the type system infers their types according to how they are used in *lseek*. To create an encrypted stream class, one must apply the `Encrypted` mixin to an existing stream class. For example, `Encrypted` $\diamond$ `FileStream` is an encrypted file class. Note that mixin `Encrypted` can be applied to a family of different streams. For example, we can construct `Encrypted` $\diamond$ `Socket`, which is a class that encrypts data communicated over a network. Moreover, we can express many uses of multiple inheritance by applying more than one mixin to a class. For example, `PGPSign` $\diamond$ `UUEncode` $\diamond$ `Encrypted` $\diamond$ `Compress` $\diamond$ `FileStream` produces a class of files that are compressed, then encrypted, then uuencoded, then signed. Furthermore, mixins can be used for some other forms of inheritance. In the above example, the result of applying `Encrypted` to a stream satisfies the constraint required by `Encrypted` itself, therefore, we can apply `Encrypted` more than once: `Encrypted` $\diamond$ `Encrypted` $\diamond$ `FileStream` is a class of files that are encrypted twice. In our system, class private fields do not conflict even if they have the same name, so each application of `Encrypted` can have its own encryption key.

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

```
let m1 = build_compression() in let m2 = build_encryption() in
  let m = m1 • m2 in (new(m ◇ FileStream)).write("foo")
```

The function `build_compression` returns a specific mixin according to user's requests: it can return a simple `Compress` mixin, or a more elaborate `UUEncode` $\bullet$ `Compress` mixin. Similarly, `build_encryption`, instead of simply returning a mixin `Encrypted`, returns the composition `PGPSign` $\bullet$ `Encrypted`. All these modular functionalities would not be *directly* provided by simple mixin application. In fact, if from an algebraic point of view it is desirable that $(M_1 \bullet M_2) \diamond C = M_1 \diamond (M_2 \diamond C)$ (this holds in our calculus), it is also true that the ability of composing $(M_1 \bullet M_2)$ without creating immediately a class, therefore of keeping the composition as an "incomplete class", gives the possibility of using it within all wanted contexts without rewriting the mixins code.

4. REFERENCES

[1] 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.

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

[3] 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.

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

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

[6] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.