Mixins in Strongtalk

Lars Bak
Gilad Bracha
Steffen Grarup
Robert Griesemer
David Griswold
Urs Hölzle

June 6, 2002

Abstract

We describe the use and implementation of mixins [BC90] in the Animorphic Smalltalk system, a high performance Smalltalk virtual machine and programming environment. Mixins are the basic unit of implementation, and are directly supported by the VM. At the language level, code can be defined either in mixins or in classes, but classes are merely sugar for mixin definition and application. The Strongtalk type system supports the optional static typechecking of mixins in an encapsulated manner. Independent of typechecking, the resulting system system substantially outperforms existing Smalltalk implementations.

1 Introduction

The concept of a mixin originates in the LISP community [Moo86], where it referred to a class designed to operate with a variety of superclasses. LISP mixins were classes that observed a programming convention that leveraged off the Flavors/CLOS multiple inheritance linearization algorithms. Mixins were not a language construct in Flavors or CLOS.

In [BC90] mixins were identified as a formal linguistic construct. In this paper, we use the term mixin in the sense of [BC90] unless otherwise stated. A mixin is an abstract subclass parameterized by its superclass.

Here we report on the design and implementation of mixins in the context of a high performance Smalltalk, the Animorphic Smalltalk system. While various

Presentation at the ECOOP 2002 Inheritance Workshop.

authors have studied mixins from a programming language point of view, the work reported here is distinguished by:

- A high performance implementation.
- A complete reflective API and programming environment, including an optional, incremental typechecker.

In the Animorphic VM, mixins rather than classes are the basic unit of implementation. This carries no performance penalty; on the contrary, the Animorphic VM outperforms other Smalltalk implemementations by a substantial factor.

At the language level, code can be defined either in mixins or in classes. However, all program code is stored in mixins internally. A class is simply sugar for a corresponding mixin definition and the application of that mixin to a particular superclass.

The system includes the optional Strongtalk type system that supports the static typechecking of mixin definition and use in an encapsulated manner.

Smalltalk implementations usually comprise not only a run-time system, but a class library and interactive development environment (IDE). This is true of the Animorphic system as well. A complete blue-book [GR83] class library is included. Mixins are used at key points in the library and the programming environment supports the browsing of mixin declarations. Underlying the browsers is a reflective API that provides access to mixins. Indeed, mixins are the basic structure manipulated by this API.

The rest of the paper is structured as follows:

Section 2 discusses the basic programming model. Section 3 shows examples of mixin usage. Section 4 discusses the (optional) typechecking of mixins. Section 5 discusses interactions between mixins and reflection. Section 6 discusses the implementation. Section 7 discusses trade offs in language design and compares our approach to others. Section 8 briefly discusses the project's status and history. Finally we discuss our contributions and draw conclusions.

2 Basic Model

A mixin specifies a set of modifications (overrides and/or extensions) to be applied to a superclass parameter. A mixin differs from an ordinary subclass definition in that it abstracts over the identity of its superclass.

Mathematically, a mixin is a function that maps a class S to a subclass of S with a particular class body.

Consider the following example [1]:

```
mixin Comparable(G) {
```

[1] Smalltalk programmers will recognize this code is based on the standard library class Magnitude. In the Animorphic system, Magnitude is defined as a class for compatibility, and then used as a mixin as explained below.

```
G subclass {
≤ that
  ^ (self == that) || (self < that)
> that
  ^ that < self
≥ that
  ^ that ≤ self
}}
```

The mixin Comparable abstracts over an unspecified *formal superclass* G. The code in the mixin assumes that the G implements a boolean valued method $<$ that implements the less-than relation. Comparable can be used at different points in the subclass hierarchy, by invoking it on different actual superclasses, much as one invokes a function at multiple points in a computation. This is accomplished using $\triangleright$, the mixin invocation operator :

ComparablePoint = Comparable $\triangleright$ Point
Number = Comparable $\triangleright$ BasicNumber

The $\triangleright$ operator takes a mixin M and a (super)class S, and produces a new class that modifies its superclass S with the code defined in M. The mixin can be invoked on various superclasses to derive different classes. We refer to the resulting classes as *mixin invocations.* In all cases the source code of the mixin is shared among all these mixin invocations, promoting modularity.

The view of mixins as functions is useful in understanding the properties of mixins in our model. Here are some key points:

- We distinguish between mixins and classes, just as one distinguishes between functions and the values they take as arguments and produce as results. A mixin is not a class; it must be invoked on an actual superclass parameter to produce a class.

- Mixins do not affect the semantics of subclass construction or method lookup. This should be expected, as mixins result from the direct application of the principle of procedural abstraction to subclassing.

- A mixin may not be applicable to all classes, much as a function is not necessarily defined for all inputs. A function may place specific requirements on its actual parameters. Likewise, a mixin may place various requirements on the superclass. For example, a mixin may contain calls to super methods. These must be defined for any actual superclass. The precise constraints will depend on the subclassing rules of the language. Some of the requirements a mixin imposes on its actual superclasses may be expressed via type annotations. See section 4 for further discussion.

- A mixin can be composed with another mixin, to produce a *composite mixin* , in a manner completely analogous to function composition. We define *mixin composition* as follows: $(M_1 * M_2) \triangleright S = M_1 \triangleright (M_2 \triangleright S)$.

Every class implicitly defines a mixin whose body is identical to the class body. In our system, we recognize this and allow mixins to be derived from classes. A *mixin derivation,* written C mixin, denotes the mixin implicitly defined by the class C.

Here is an example taken from the Animorphic UI library. Assume the class Region represents a region on the screen. It is natural to have a class that groups several Regions together into a single composite Region:

Region subclass CompositeRegion...

It is useful to think of this class as a collection of Regions. However, one cannot usually inherit functionality from Collection since graphical widgets belong to a separate hierarchy.

In order to reuse all of the functionality in the Collection class, we revise the definition of CompositeRegion as follows:

(Collection mixin ▷ Region) subclass CompositeRegion...

Now, the superclass of CompositeRegion is a *mixin invocation* Collection mixin ▷ Region. The mixin being invoked here is a mixin derivation Collection mixin. The derived mixin operates exactly as if the functionality of the class Collection had instead been given as an explicit mixin declaration.

Programmers may define code in the context of either a class or a mixin. Defining code in classes is often advantageous because it is a familiar paradigm, and because it is relatively concrete. On the other hand, defining mixins supports a focus on reuse.

In either case, the ability to use the new code of a class in other class hierarchies is unaffected.

3 Examples of Usage

3.1 Example 1: Critical Section

Our first example is shown in figure 1. The figure is a screen shot of a mixin browser on the mixin InstanceCritical. The InstanceCritical mixin expresses the functionality of a critical section (monitor) . It has an instance variable, monitor, representing the semaphore, with an associated access method. The mixin includes a method critical: that accepts a closure as its argument and executes it within the monitor.

The browser shows Strongtalk type signatures, enclosed in angle brackets, for the instance variables and methods. In addition, in order to typecheck the mixin declaration type information regarding potential superclasses is required (see section 4.3.1 for further details). In this case, the mixin is applicable to any class, that is, any subclass of Object. This is reflected at the top of the browser window in the heading **Mixin on Object**.

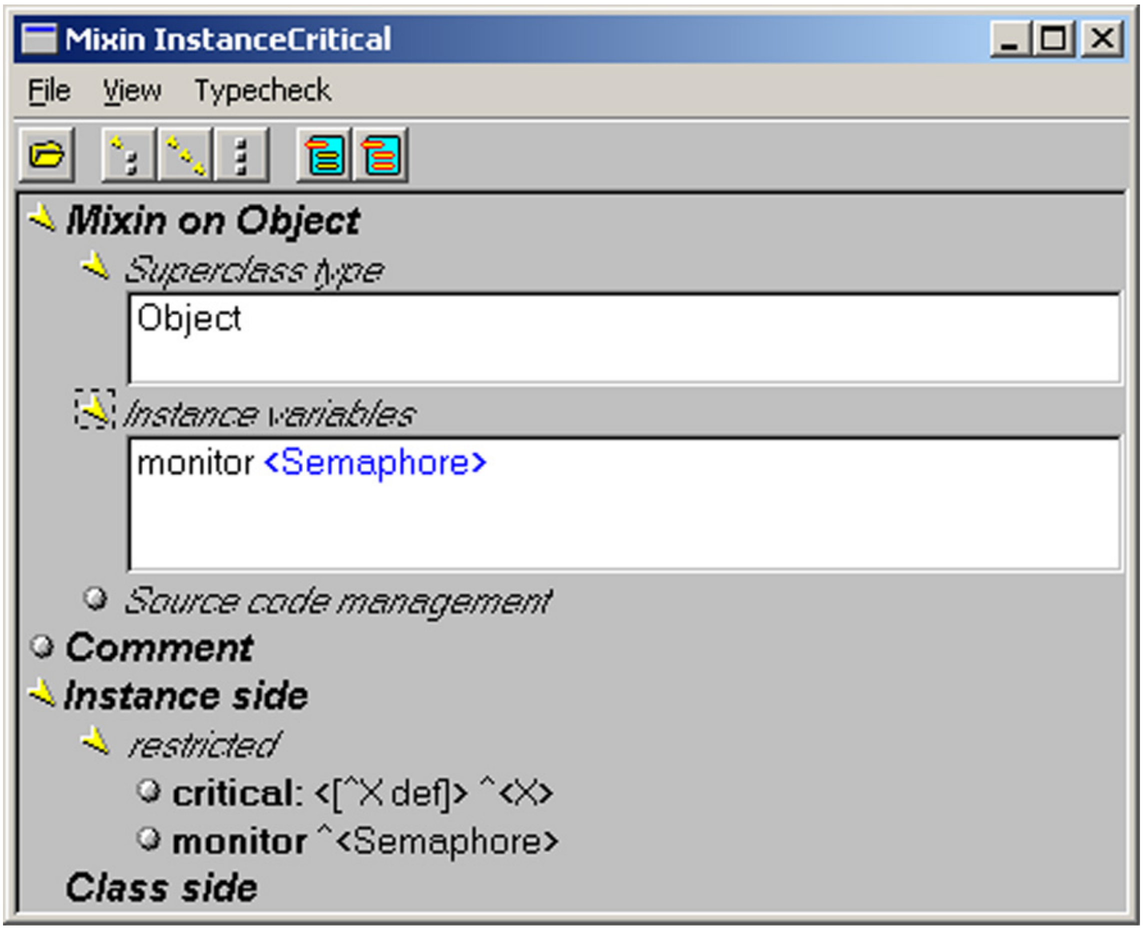

Figure 1: A mixin implementing a monitor

Without mixins, one might have to duplicate this functionality wherever it was needed. Alternately, one could add this to class Object. However, without specialized language support (e.g., as in the Java programming language [GJSB00]), significant overhead is added to every object. Note that this design does not preclude a similar optimization.

3.2 Example 2: I/O Streams

Consider the case of a stream that can perform both input and output. It combines the functionality of an input stream and an output stream. Should InputOutputStream be in the InputStream hierarchy or the OutputStreamHierarchy? In either case, we will have to duplicate functionality from one of the stream classes. We can avoid this problem if we use mixins for the input and output streams.

The actual declaration of an I/O stream in the Animorphic system is

BasicOutputStream mixin ▷ BasicReadStream subclass BasicReadWriteStream

4 Typechecking

Mixins interact with the type system in several ways. Mixin declarations and invocations must be typechecked. In addition, in a nominal type system, mixin declarations affect the subtype relation. We now address each of these issues in turn.

It is crucial that once a mixin declaration M has been type checked, invocations of the mixin can be checked on the basis of the mixin's interface, without recourse to the mixin's internals. In order to achieve this, it is important to understand what the interface of a mixin is; this in turn depends on an understanding of the concepts of *mixin signature* and *class signature*.

4.1 The Signature of a Class

Typechecking a class generally requires a comparison of the types of members of the class to the types of corresponding members in the superclass. In type systems that allow for typechecking a class without access to the source code of the superclass, there is always a (usually implicit) *class signature* that provides the information necessary to verify the typing constraints when inheriting from a class.

Animorphic Smalltalk uses the Strongtalk [BG93, Bra96] type system. Types in Strongtalk never expose private methods or variables of any kind. In contrast, a Strongtalk class signature includes the signatures and visibilities of all methods and messages, and the types of all instance and class variables. The class signature includes all inherited members. It follows that the class signature of a class C is entirely different from both the type of an instance of C, and from the type C *class*, the type of the unique instance of C's metaclass.

4.2 The Signature of a Mixin

A mixin signature encapsulates type information about the mixin, which is needed to check its application to a concrete superclass. This amounts to the declared signature of the formal superclass and types of all fields and methods declared within the mixin itself.

A mixin that requires superclasses that conform to class signature S has S as its domain. If, given a class with signature S, the mixin produces a class of signature C, then C is the range of the mixin. The type of a mixin with domain S and range C can be written as a function type on class signatures: $S \to C$. However, it is more convenient to represent it as pair (S, δ). Here, S is the signature of the superclass, as before. δ is a class signature that only includes information about the members declared by the mixin itself.

4.3 Typechecking a Mixin

Having reviewed the concepts of class signature and mixin signature, we now proceed with a discussion of mixin typechecking. Mixin typechecking can be further subdivided into two parts: typechecking mixin declarations and typechecking mixin invocations.

4.3.1 Typechecking Mixin Declarations

In order to typecheck a mixin declaration in isolation, it is necessary for the mixin to declare the expected class signature of potential superclasses. In practice, this is done by naming a particular class. In figure 1, the class Object is used for this purpose.

Any actual superclass should conform to the signature of the named class. This does not imply that actual superclasses need to be subclasses of the class named in the superclass signature declaration.

The body of the mixin is then checked under the assumption that the superclass has the declared superclass signature. This allows the typechecker to verify that:

1. No method in the mixin has a signature that contradicts that of the declared superclass signature. For example, if a method declared in the mixin would override a method in the superclass signature, we would require that its signature be a subtype of the signature of the overridden method.

2. No field in the mixin has a signature that contradicts that of the declared superclass signature. In Smalltalk, fields are exposed to subclasses, and so a mixin may not declare a field of the same name as a field declared in the superclass signature.

3. All super calls in the mixin are well typed (i.e., the superclass signature supports all these calls with the correct method signature).

4. All self calls in the mixin are well typed (i.e., such calls are supported either by methods declared by the mixin itself or else the superclass signature supports them with the correct method signature).

Note that all of the above checks rely on a declaration of the intended signature of the superclass. The first two items above check the relation between the mixin and its superclass parameter. The latter two items check the body of the mixin itself.

However, this in itself is not sufficient because of the "negative information problem" [CM89]. The actual superclass might include additional fields or methods. As an example, consider the class PatientStatus sketched below:

```
Object subclass PatientStatus {
...
critical: b ^ <Boolean> {...}
}
```

PatientStatus is intended to represent monitoring data for hospital patients. For our current purposes, the only relevant properties of PatientStatus are that it is a subclass of Object and that it has a method critical: that takes a boolean argument. If the argument is true, the patient's condition is regarded as critical, requiring intensive care. Now, consider the mixin invocation

InstanceCritical $\triangleright$ PatientStatus

Even though we checked that InstanceCritical is well formed assuming a superclass with signature Object, and PatientStatus is a subtype of Object, the invocation is not type correct. PatientStatus defines the method critical: that is also defined by InstanceCritical, but the argument types of InstanceCritical>>critical: and PatientStatus>>critical: are incompatible.

Evidently, additional type checks must be made at mixin invocation time, as discussed below. However, these checks do not require access to the source code for either the mixin or the actual superclass. All necessary information is provided by the signature of the actual superclass and by the mixin's signature.

4.3.2 Typechecking Mixin Invocations

At mixin invocation time, we need to verify that the actual superclass does not introduce members whose type conflicts with any corresponding members of the mixin. This is in fact a repetition of checks 1 and 2 of the previous section, applied to the *actual superclass*.

4.3.3 Typechecking Mixin Derivations and Compositions

Mixin derivations require no special typechecking. The signature of the derived mixin must be inferred. Given a class C with superclass S, the mixin type is (S, δ_C), where δ_C can be computed as the difference between the signatures of C and S.

We typecheck the mixin composition $M1 * M2$ just as we would the mixin invocation $M1 \triangleright (M2 \triangleright S)$, where S is the default superclass of $M2$.

4.4 Interactions with Subtyping

4.4.1 Motivation

When subtyping is structural, mixins do not introduce any new issues with respect to subtyping. However, most practical programming languages use nominal subtyping, where subtyping is based upon explicitly declared relationships. Strongtalk started out as a structural type system and evolved into a nominal one over time [2]. Special care must be taken to support the use of mixins in a nominal type system.

The basic subtyping rule used in class-based languages with nominal subtyping systems is

[2] The reasons for this evolution are outside the scope of this paper.

$C_1 \leq C_2$ iff either

1. $C_1 == C_2$
2. C_1 is a direct subclass of S and $S \leq C_2$.

where C_1 and C_2 are classes.

Consider the class CompositeRegion shown earlier. Recall that CompositeRegion was defined as

(Collection mixin $\triangleright$ Region) subclass CompositeRegion...

The above rule allows us to conclude that CompositeRegion $\leq$ Region but not that CompositeRegion $\leq$ Collection. Clearly, some extension of the usual rules of nominal subtyping is called for.

4.4.2 Mixin Subtyping in Strongtalk

In Strongtalk, every class declaration and *every mixin declaration* implicitly introduces a type of the same name as the declaration. The type implicitly defined for a class is known as a *protocol* which consists of a set of message signatures; it is similar to an interface in the Java programming language. The protocol of a class is a direct subtype of the protocol of its superclass, extended by those public message signatures declared by the class itself.

The type implicitly defined for a mixin is similar, but mixins don't have a superclass. Instead, we use the protocol of the class used to declare the expected superclass signature as the supertype.

A mixin invocation $M \triangleright S$ has as its implicit type a *mixin type* of the same name, written $M \triangleright S$. For purposes of subtyping, a mixin type $t_1 \triangleright t_2$ acts like an intersection type. Here are the subtyping rules for mixin types:

1. $t \leq s_1 \triangleright s_2$ if $t \leq s_1$ and $t \leq s_2$.
2. $t_1 \triangleright t_2 \leq s$ if $t_1 \leq s$ or $t_2 \leq s$.

The intuition for the first rule is this: the set of message signatures supported by $s_1 \triangleright s_2$ consists of those signatures that are supported by s_1, augmented by any additional signatures inherited from s_2. $s_1 \triangleright s_2$ does not support any message signature not provided by either s_1 or s_2. Since $t \leq s_1$ and $t \leq s_2$, t must support all signatures in $s_1 \triangleright s_2$, and is therefore a structural subtype of $s_1 \triangleright s_2$.

The second rule can be justified intuitively as follows: A mixin invocation $t_1 \triangleright t_2$ must be a subtype of t_2 - this is ensured by the rules for typechecking mixin invocations given in section 4.3.2. By transitivity, $t_1 \triangleright t_2 \leq s$ if $t_2 \leq s$. More subtly $t_1 \triangleright t_2$ is also a structural subtype of t_1. To see that, note that:

- t_1 must have been declared with an expected superclass signature C.

- Every message signature declared in t_1 must be supported by $t_1 \rhd t_2$ - it could not be overridden by t_2.
- Any other message signature supported by t_1 must have been inherited from C. But the typechecking requirements given in 4.3.2 above ensure that $t_2 \leq C$. Thus, any message signatures inherited by t_1 from C (or subtypes thereof) will also be inherited from t_2 to $t_1 \rhd t_2$

$t_1 \rhd t_2 \leq s$ if $t_1 \leq s$ or $t_2 \leq s$ follows directly.
Similar rules are given for gbeta [Ern99] and Scala [Ode01].

5 Mixins and Reflection

5.1 Mixins as Objects

Mixins are reified as objects in Animorphic Smalltalk. Every class is an invocation of some mixin, which can be obtained via the system's reflective interface. Using reflection, one can determine the structure of a particular mixin (e.g., what instance variables or methods it defines). One can also find out what invocations of the mixin exist in the system, as exemplified by the following code fragment, which prints a list of all classes that invoke the monitor abstraction shown earlier:

```
(Mirror on: InstanceCritical) invocations do:
    [:inv | Transcript show: inv name; cr]
```

Mirrors are special objects that are used to reflect other objects [UCCH90]. Having obtained a mirror on InstanceCritical, we can then obtain a list of invocations and execute a closure for each element in the list. The code in the closure simply prints the line with the name of an invocation on the system transcript.

5.2 Reflective Update

Mixins are the unit of reflective update, because they are the unit of code storage in the VM. In order to perform a reflective change to a class, a copy of the class' mixin is first made. The copy is then changed as desired. Finally, the copy is *installed* atomically in place of the original mixin. At this point, all invocations of this mixin, their subclasses and the instances of all these classes are updated to reflect the changes made to the mixin.

Mixins are in some respects easier to use for reflective update than classes. A mixin cannot have any instances and a mixin object has no methods that contain user code. There is nothing we can do with an uninstalled mixin except reflect it or submit it for installation. As a result, a series of reflective changes can be made on an uninstalled mixin without the need to update the entire system at each individual change. It is even possible to simultaneously install multiple mixins in a single atomic operation.

This has two advantages.

1. A series of reflective changes can constitute a single transaction: none of the changes will be made unless all of them succeed, and they will all take place at once. Many program changes fit this pattern.

2. If several schema changes are made, it is much cheaper to batch the changes and avoid repeated traversals of the heap to update all instances.

This contrasts with the standard approach taken in Smalltalk where each change operation takes immediate effect.

It is more difficult to obtain the desired properties 1 and 2 if reflective changes are made on classes rather than mixins. To see why, we will examine two alternatives: *scratchpad classes* and *functional update*.

One might consider making copies of classes to be used as "scratchpads", with the intent of changing their structure and installing them later. This is analogous to the approach we take with mixins, but it fails with classes for the following reasons:

- Classes may be instantiated. Imagine that a scratchpad class is instantiated, and then subjected to a series of change operations, such as adding instance variables and methods that access them. If changes to a scratchpad class do not take immediate effect, an inconsistency arises between the class and its instances. Attempting to invoke a method on such an instance could lead to undefined behavior.

- Classes are usually stateful. In Smalltalk, classes may have their own instance variables as well as class variables. Similar considerations apply in other languages. Eiffel [Mey88] has `once` per-class variables. In Java classes are associated with static variables and per-class locks etc.

  Once a scratchpad copy of a class has been created the state of the original class may evolve independently of the state of the scratchpad class. When a scratchpad class is installed, we have to somehow reconcile its state with that of the class being updated. Do we preserve its state, or the state of the class being updated, or some combination of the two?

- Classes have application specific methods (class methods). This is true not only in Smalltalk, but in Java as well. If these methods can be invoked before the class is installed, the notion of a transactional change is lost.

To avoid these problems we would have to prevent such uninstalled classes from being instantiated or accepting application specific messages of any kind. These restrictions effectively turn uninstalled classes into a completely different entity than a normal class. They effectively become *class descriptions* that are independent of the class from which they are derived. Such descriptions do not interact with non-reflective program state in any way. These descriptions are very much like mixins, except that they specify a particular superclass.

An alternative approach is functional update. Reflective changes on a class would not destructively alter the class. Instead, each reflective change operation would return a fresh copy of the class, with the appropriate changes.

This solution suffers from several disadvantages.

- The copying involved may be costly.
- The copied classes must have distinct names which must be managed automatically.
- Invocation of class methods could still disturb invariants of the original class.

Our conclusion is that any solution should perform reflective changes on a purely declarative description. In practice, mixins give us such a description.

6 Implementation

In this section, we discuss the implementation of mixins in the context of the Animorphic VM. We will not discuss the Animorphic VM's overall implementation in any detail. Instead, we give a brief overview to provide context, and focus on the details of mixins.

6.1 Highlights of the Animorphic VM

The Animorphic VM combines a high-performance interpreter with a dynamic compiler into a mixed-mode execution engine. Invocation of the dynamic compiler is determined based upon dynamically-obtained profile data [Höl94, Mic].

Dynamic method dispatch is supported via the use of polymorphic inline caches (PICs) [HCU91]. Classes and mixins have associated method tables that store references to their methods, but these must be distinguished from virtual dispatch tables commonly used in the implementation of languages such as C++ and Java. No virtual dispatch tables are used in the Animorphic system.

6.2 Transparent Lookup

In a mixin based system, a class does not include all of the methods it declares directly in its method table. These methods are naturally shared in the mixin's method table. Consequently, the method lookup process must be changed to consult the mixin. However, these changes can in principle be localized to the routine that scans the class hierarchy during method lookup. Changes to lookup can be *transparent* to the rest of the VM. In particular, caching techniques such as PICs etc. operate unchanged.

In reality, VMs operate on concrete data structures and perform various performance optimizations that complicate this picture.

6.3 Data Structure

The VM represents mixins as special objects, that contain the description of a class delta. The mixin contains descriptions of the instance variables and class variables, and a method dictionary where all code is initially stored.

All mixin invocations contain the class variables (which are distinct for each invocation, though shared by an invocation and all its subclasses), the class' instance variables (which are distinct for every invocation, and indeed for every class), the superclass (which is unique to each invocation), a pointer to the mixin, and additional implementation information (such as the offset of the instance variables, which varies depending on the superclass).

One of the problems when sharing code among mixin invocations is that the physical layout of instances varies between invocations. In high performance code, references to instance variables must be converted into physical offsets relative to the beginning of the instance. This is not possible if the code is shared among invocations with different instance structure. This problem is addressed via the *copying down* mechanism, described below.

6.4 Copying down Methods

Methods that do not access instance variables or `super` are shared in the mixin. Methods that access instance variables may have to be specialized for the invocation, where the instance variable access is *customized* according to the structure of instances of the invocation.

All customization means in this case, is asking the compiler to compile a method in the context of a particular class - the mixin invocation in question. The customized version must be installed into the invocation. We refer to the process of customization and installation as *copying down*.

Rather than copy down methods when a new mixin invocation is created, we copy down methods into invocations lazily, the first time a method is invoked.

The mixin's method table contains a version of every method defined in the mixin, even methods that access instance variables or `super`. This version must always exist, as a template that mixin invocations will use to customize the method when necessary.

When a method is called on a mixin invocation it is looked up, and if no customized version exists, the version in the mixin is found. It can then be customized for the invocation in question, and this customized version is installed into the invocation.

Copying down can be avoided entirely in certain circumstances:

- For explicitly declared mixins (as opposed to mixin derivations and compositions), methods that access instance variables are compiled assuming an initial offset of 0. When an invocation's actual superclass has no instance variables the version compiled for the mixin can be used and no copying down is needed.

- If a mixin represents a class declaration, then we associate the mixin with its *master invocation*, which is the class from which the mixin is derived. The master invocation is stored in an instance variable of the mixin. Any invocation can check if it is the master by examining its mixin and seeing if the master is identical to itself.

  We customize the method to the master invocation in place, when the method is first called. Thus, the master invocation does not need to have any methods copied down. When a method is invoked on another invocation, copying down can also be avoided if the size of instances of the invocation's superclass is the same as that of instances of the master invocation's superclass.

Methods that access `super` must be copied down except in the case of the master invocation.

6.5 The Cost of Mixins

The high performance of the Animorphic VM is independent of the use of mixins. It is not achieved due to the use of mixins. At the same time, mixins do not degrade performance. Mixins have essentially no performance impact whatsoever.

There is no per-method space overhead for using mixins as long as they are derived from classes in the conventional Smalltalk style. A per method space overhead may occur only for some methods in regular mixins, which by design are likely to have multiple invocations. In practice, general purpose mixins rarely have state or access `super` so this occurence is infrequent.

We only have the slight per-class overhead, which is negligible, and the run time overhead of lazy customization, which is also negligible, since it only happens at the first call. The reflective API and classes used to manage mixins add a small fixed cost as well.

Mixins do tend to increase the degree of polymorphism in the system. One can expect to see the same performance effects as in any other highly polymorphic code.

7 Related Work

A very brief description of the Animorphic system's mixins was given in [BG96]. The mixins implemented in the Animorphic system are based on the semantics given in [BC90]. Alternative models of mixins that attempt to address some of the weaknesses of this model have been proposed. The first was the notion of modules proposed in [Bra92]. Since that early work, there has been a considerable amount of research on mixins.

Gbeta [Ern99, Ern01] unifies Beta patterns with mixins. However, Gbeta incorporates a mechanism for automated linearization of mixins unlike the model of [BC90] which we use. Gbeta is a statically typed language, and typechecks mixin declarations and invocations in a manner similar to our system.

The unification of mixins and classes in gbeta demonstrates an important point: using mixins, one can avoid the need for a distinction between types and implementations (e.g., classes and interfaces in Java, classes/mixins and protocols in Strongtalk) while retaining the advantages of multiple classification that are the primary advantage of this distinction.

The Scala language [Ode01] supports mixins along a model similar to ours, with a related (but mandatory) type discipline. Scala's treatment of mixins differs in several respects. Mixins are never explicitly declared in Scala. Instead, they are implicitly derived from class definitions. In the context of creating a particular instance, a class' superclass can be replaced, implicitly converting the class into a mixin. This approach also cleanly deals with the notion of constructors, which can be problematic when combined with mixins (see [ALZ00]). However, the Scala implementation differs substantially from our work, because Scala is implemented by translation to Java virtual machine byte codes.

In Ruby [Mat01, TH01], mixins are defined as modules, and these may then be included in other modules or in classes. Despite the considerable difference in surface syntax, the underlying model of mixins and classes used in Ruby is essentially the same as in Animorphic Smalltalk. The Ruby library makes significant use of mixins, much as the Animorphic library does. Ruby does not provide a type system or a high performance implementation of mixins.

Ancona et al. present JAM, an extension of the Java programming language with mixins in [ALZ00]. The mixins they present are conceptually similar to the ones discussed here. They discuss typechecking mixins along principles similar to those presented here. However, they do not deal with mixin composition or with automatic derivation of mixins from classes. Their work focuses on Java-specific complications and limitations.

There have been several other efforts to typecheck mixins. The essential intuition for typechecking mixins was sketched in chapter 3 of [Bra92]. Formal type systems for typechecking mixins are given in [FKF98, BPS99] as well as in [ALZ00].

Our work differs from these in the following respects:

- It is part of an optional type system.
- It is implemented as part of an interactive and incremental typechecker in the context of a complete IDE
- We do not have a comprehensive formalization.
- We give an account of typechecking mixin composition.

Various researchers have explored mixin-like constructs with somewhat different semantics.

Smaragdakis and Batory [SB98] discuss *mixin layers*, a mechanism based on nested mixins, implemented using C++ templates.

Steyaert et al. [SCD+93] studied the use of *mixin-methods*, a mechanism designed to allow classes to control what extensions might be applied to themselves.

Duggan [Dug96] has proposed mixin modules, an extension of the ML module system designed to support mutually recursive modules. Mixin modules may be composed into ordinary modules, in a manner related to the creation of classes by mixins. However, ML and its module system are very different from most object oriented languages, and exploring the relationship is far beyond the scope of this paper. In later work [Dug00], Duggan has applied the principles of mixin based inheritance to the construction of composable language interpreters.

Several research proposals have introduced variations on the notion of mixins that are concerned with the potential problem of inadvertant overrides. When a mixin is defined, the programmer does not know the concrete superclass. Consequently, the mixin may declare methods that unintentionally override methods of particular superclasses.

In the untyped model defined here, there is no way to distinguish between methods that are intended to override superclass methods and other methods of a mixin. Using types, it is possible to warn of unintended overrides. Nevertheless, the overrides occur. This might be problematic, though there is little practical experience to demonstrate the problem conclusively.

In response, Flatt et al. [FKF98, FF98b, FF98a] have proposed alternate semantics for mixins. In their approach, mixins specify what interfaces they intend to override. Methods of a specified interface are overridden only by corresponding methods of the same interface.

Similarly, in Extended Moby [FR00], classes only override the known methods of their superclasses. Mixins are represented using genericity. This approach obviates the unintended override problem, and allows for typechecking a mixin without use of mixin signatures. Moby effectively allows mixin composition. However, in Moby it is not possible to derive a mixin from a class after the fact as in our system. The Moby approach relies on the presence of a mandated static type system. Furthermore, it is predicated on abandoning subsumption.

An alternative proposed by Bono [BPS99] is to distinguish overriding methods syntactically.

None of the proposals that attempt to address the overriding problem discuss implementation techniques for mixins in any detail. However, these approaches lend themselves more readily to dispatch table based implementations. Lookup based implementations may require the use of name mangling schemes.

The approaches above involve more elaborate semantics than those used in our system, motivated by the perceived problem of inadvertant overrides. We have opted for simpler semantics, because:

- They are better suited to a dynamically typed (or optionally typed) language.

- We have not observed the problem that these richer constructs attempt to address in practice.
- They are simpler and therefore more understandable.

8 Status/History

The system described in this paper was developed starting in the fall of 1994 as part of the Animorphic project. The Animorphic project produced a working high performance Smalltalk system, including VM, blue book library, UI framework and incremental development environment. In late summer of 1996, all work on Animorphic Smalltalk effectively ceased due to commercial considerations that dictated an emphasis on Java virtual machines. As of July 2002, a public release of Animorphic Smalltalk will be available for download.

9 Conclusions

We have presented a working high-performance implementation of mixins that adheres to the well defined semantic model given [BC90]. We have shown that mixins can be implemented efficiently, incurring negligible overhead in time and space when compared to a high-performance, class based implementation of a dynamically typed language.

We have also shown how to optionally statically typecheck such mixins in the context of a nominal type system, how to incorporate them in a reflective architecture, and how they may be useful in realistic applications such as a working UI framework and in IO libraries.

Acknowledgements

The Animorphic system could not have been built without the inspiration of David Ungar and Randy Smith's groundbreaking work on Self [US87].

The public release of the system owes much to the initiative of Eliot Miranda. At Sun Microsystems, several individuals contributed to the decision to release the system: Larry Abrahams, Rich Green, and especially Richard Gabriel.

References

[ALZ00] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - a smooth extension of Java with mixins. In *European Conference on Object-Oriented Programming*, pages 154–178, 2000.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, September 1993.

[BG96] Gilad Bracha and David Griswold. Extending Smalltalk with mixins, September 1996. OOPSLA Workshop on Extending the Smalltalk Language.

[BPS99] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 43–66, Lisbon, Portugal, June 1999. Springer-Verlag.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[Bra96] Gilad Bracha. The Strongtalk type system for Smalltalk, September 1996. OOPSLA Workshop on Extending the Smalltalk Language.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. Technical Report 48, Digital Equipment Corporation Systems Research Center, August 1989.

[Dug96] Dominic Duggan. Mixin modules. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, 1996.

[Dug00] Dominic Duggan. A mixin-based, semantics-based approach to reusing domain-specific programming languages. In *European Conference on Object-Oriented Programming*, pages 179–200, 2000.

[Ern99] Erik Ernst. Propagating class and method combination. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 67–91, Lisbon, Portugal, June 1999. Springer-Verlag.

[Ern01] Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming*, pages 303–326, 2001.

[FF98a] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[FF98b] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 171–183, 1998.

[FR00] Kathleen Fisher and John Reppy. Extending Moby with inheritance-based subtyping. In *European Conference on Object-Oriented Programming*, pages 83–107, 2000.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, Reading, Massachusetts, 2000.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic in-line caches. In P. America, editor, *Proceedings ECOOP'91*, LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[Höl94] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Department of Computer Science, Stanford University, 1994. Available at http://www.cs.ucsb.edu/labs/oocsb/papers/urs-thesis.html.

[Mat01] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly & Associates, 2001.

[Mey88] Bertrand Meyer. *Object Oriented Software Construction*. Prentice-Hall International, Hertfordshire, England, 1988.

[Mic] Sun Microsystems. The Java Hotspot^{tm} virtual machine. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot _WP_Final_4_30_01.html.

[Moo86] David A. Moon. Object-oriented programming with Flavors. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 1–8, 1986.

[Ode01] Martin Odersky. Report on the programming language Scala, 2001. available at http://lampwww.epfl.ch/ odersky/scala/.

[SB98] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixins layers. In *European Conference on Object-Oriented Programming*, pages 550–570, 1998.

[SCD+93] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested mixin-methods in agora. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 197–219, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[TH01] David Thomas and Andrew Hunt. *Programming Ruby: A Pragmatic Programmer's Guide.* Addison-Wesley, 2001.

[UCCH90] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. The SELF manual, version 1.0, July 1990.

[US87] David Ungar and Randall Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1987.