Classes and Mixins

Matthew Flatt  Shriram Krishnamurthi  Matthias Felleisen
Department of Computer Science
Rice University
Houston, Texas 77005-1892

Abstract

While class-based object-oriented programming languages provide a flexible mechanism for re-using and managing related pieces of code, they typically lack linguistic facilities for specifying a uniform extension of many classes with one set of fields and methods. As a result, programmers are unable to express certain abstractions over classes.

In this paper we develop a model of class-to-class functions that we refer to as mixins. A mixin function maps a class to an extended class by adding or overriding fields and methods. Programming with mixins is similar to programming with single inheritance classes, but mixins more directly encourage programming to interfaces.

The paper develops these ideas within the context of Java. The results are:

1. an intuitive model of an essential Java subset;
2. an extension that explains and models mixins; and
3. type soundness theorems for these languages.

1 Organizing Programs with Functions and Classes

Object-oriented programming languages offer classes, inheritance, and overriding to parameterize over program pieces for management purposes and re-use. Functional programming languages provide various flavors of functional abstractions for the same purpose. The latter model was developed from a well-known, highly developed mathematical theory. The former grew in response to the need to manage large programs and to re-use as many components as possible.

Each form of parameterization is useful for certain situations. With higher-order functions, a programmer can easily define many functions that share a similar core but differ in a few details. As many language designers and programmers readily acknowledge, however, the functional approach to parameterization is best used in situations with a relatively small number of parameters. When a function must consume a large number of arguments, the approach quickly becomes unwieldy, especially if many of the arguments are the same for most of the function’s uses.

Class systems provide a simple and flexible mechanism for managing collections of highly parameterized program pieces. Using class extension (inheritance) and overriding, a programmer derives a new class by specifying only the elements that change in the derived class. Nevertheless, a pure class-based approach suffers from a lack of abstractions that specify uniform extensions and modifications of classes. For example, the construction of a programming environment may require many kinds of text editor frames, including frames that can contain multiple text buffers and frames that support searching. In Java, for example, we cannot implement all combinations of multiple-buffer and searchable frames using derived classes. If we choose to define a class for all multi-buffer frames, there can be no class that includes only searchable frames. Hence, we must repeat the code that connects a frame to the search engine in at least two branches of the class hierarchy: once for single-buffer searchable frames and again for multiple-buffer searchable frames. If we could instead specify a mapping from editor frame classes to searchable editor frame classes, then the code connecting a frame to the search engine could be abstracted and maintained separately.

Some class-based object-oriented programming languages provide multiple inheritance, which permits a programmer to create a class by extending more than one class at once. A programmer who also follows a particular protocol for such extensions can mimic the use of class-to-class functions. Common Lisp programmers refer to this protocol as mixin programming [21, 22], because it roughly corresponds to mixing in additional ingredients during class creation. Bracha and Cook [6] designed a language of class manipulators that promote mixin thinking in this style and permit programmers to build mixin-like classes. Unfortunately, multiple inheritance and its cousins are semantically complex and difficult to understand for programmers. As a result, implementing a mixin protocol with these approaches is error-prone and typically avoided.

For the design of MzScheme’s class and interface system [15], we experimented with a different approach. In MzScheme, classes form a single inheritance hierarchy, but are also first-class values that can be created and extended at run-time. Once this capability was available, the program

4Function entry points à la Fortran or keyword arguments à la Common Lisp are a symptom of this problem, not a remedy.

5Don Friedman determined in an informal poll in 1990 that almost nobody who teaches C++ teaches multiple inheritance [fsm. com].
interface Place\textsuperscript{1}...
interface Barrier\textsuperscript{1}...
interface Door\textsuperscript{1} extends Place\textsuperscript{1}, Barrier\textsuperscript{1}...

class Door\textsuperscript{2} extends Object implements Door\textsuperscript{1} {
  Room\textsuperscript{2} Enter(Person\textsuperscript{2} p) { ... } ...
}
class LockedDoor\textsuperscript{2} extends Door\textsuperscript{2}...
class ShortDoor\textsuperscript{2} extends Door\textsuperscript{2}...

Figure 1: A program determines a static directed acyclic graph of types

define (player Person Room LockedDoor ShortDoor)
  door.Enter(player) <-

Figure 2: In the context of a type graph, reductions map a store-expression pair to a new store-expression pair

ners of our team used it extensively for the construction of
DrScheme [14], a Scheme programming environment. How-
ever, a thorough analysis reveals that the code only contains
first-order functions on classes.

In this paper, we present a typed model of such "class
functors" for Java [18]. We refer to the functors as mixins
due to their similarity to Common Lisp's multiple inher-
tance mechanism and Bracha's class operators. Our pro-
posal is superior in that it isolates the useful aspects of
multiple inheritance yet retains the simple, intuitive nature
of class-oriented Java programming. In the following section,
we develop a calculus of Java classes. In the third section,
we motivate mixins as an extension of classes using a small
but illuminating example. The fourth section extends the
type-theoretic model of Java to mixins. The last section
considers implementation strategies for mixins and puts our
work in perspective.

2 A Model of Classes

CLASSICJAVA is a small but essential subset of sequential
Java. To model its type structure and semantics, we use
well-known type elaboration and rewriting techniques for
Scheme and ML [13, 19, 29]. Figures 1 and 2 illustrate our
strategy. Type elaboration verifies that a program defines a
static tree of classes and a directed acyclic graph (DAG) of
interfaces. A type is simply a node in the combined graph.
Each type is annotated with its collection of fields and meth-
ods, including those inherited from its ancestors.

Evaluation is modeled as a reduction on expression-store
pairs in the context of a static type graph. Figure 2 demon-
strates reduction using a pictorial representation of the store
as a graph of objects. Each object in the store is a class-
tagged record of field values, where the tag indicates the
run-time type of the object and its field values are refer-
ences to other objects. A single reduction step may extend
the store with a new object, or it may modify a field for an
existing object in the store. Dynamic method dispatch is
accomplished by matching the class tag of an object in the
store with a node in the static class tree; a simple relation
on this tree selects an appropriate method for the dispatch.

The class model relies on an few implementation details
as possible. For example, the model defines a mathemati-
cal relation, rather than a selection algorithm, to associate
fields with classes for the purpose of type-checking and eval-
uation. Similarly, the reduction semantics only assumes that
an expression can be partitioned into a proper reduct and an
(evaluation) context; it does not provide a partitioning al-
gorithm. The model can easily be refined to expose more
implementation details [12, 19].

2.1 CLASSICJAVA Programs

The syntax for CLASSICJAVA is shown in Figure 3. A pro-
gram \( P \) is a sequence of class and interface definitions fol-
lowed by an expression. Each class definition consists of a
sequence of field declarations and a sequence of method
declarations, while an interface consists of methods only. A
method body in a class can be abstract, indicating that the
method must be overridden in a subclass before the class is
instantiated. A method body in an interface must be ab-
stract. As in Java, classes are instantiated with the new
operator, but there are no class constructors in CLASSIC-
JAVA; instance variables are always initialized to null. In the
evaluation language for CLASSICJAVA, field uses and super-
invocations are annotated by the type-checker with extra in-
formation (see the underlined parts of the syntax). Finally, the `view` and `let` forms represent Java's casting expressions and local variable bindings, respectively.

A valid CLASSICJAVA program satisfies a number of simple predicates and relations; these are described in Figure 4. For example, the predicate `ClassOnce(P)` states that each class name is defined at most once in the program `P`. The relation `<P>` associates each class name in `P` to the class it extends, and the (overloaded) `ekP` relations capture the field and method declarations of `P`.

The syntax-summarizing relations induce a second set of relations and predicates that summarize the class structure of a program. The first of these is the `subclass` relation `<P>`, which is a partial order if the `CompleteClasses(P)` and `WellFoundedClasses(P)` predicates hold. In this case, the classes declared in `P` form a tree that has `Object` at its root.

If the program describes a tree of classes, we can "decorate" each class in the tree with the collection of fields and methods that it accumulates from local declarations and inheritance. The source declaration of any field or method in a class can be computed by finding the `minimum` (i.e., farthest from the root) superclass that declares the field or method. This algorithm is described precisely by the `ekP` relations. The `ekP` relation retains information about the source class of each field, but it does not retain the source class for a method. This reflects the property of Java classes that fields cannot be overridden (so instances of a subclass always contain the field), while methods can be overridden (and may become inaccessible).

Interfaces have a similar set of relations: the super-interface declaration relation `->I` induces a sub-interface relation `<=I`. Unlike classes, a single interface can have multiple proper superinterfaces, so the sub-interface order forms a DAG instead of a tree. The methods of an interface, as described by `ekI`, are the union of the interface's declared methods and the methods of its superinterfaces.

Finally, classes and interfaces are related by `implements` declarations, as captured in the `<P>` relation. This relation is a set of edges joining the class tree and the interface graph, completing the `subtypes` picture of a program. A type in the full graph is a subtype of all of its ancestors.

2.2 CLASSICJAVA Type Elaboration

The type elaboration rules for CLASSICJAVA are defined by the following judgments:

- `P` elaborates to `P'` with type `t`\[P \vdash \Gamma \Rightarrow \text{P} : t\]
- `P` elaborates to `P'` with type `t`\
- `P` elaborates to `P'` with type `t`\
- `P` elaborates to `P'` with type `t`\

The type elaboration rules translate expressions that access a field or call a super method into annotated expressions (see the underlined parts of Figure 3). For field uses, the annotation contains the compile-time type of the instance expression, which determines the class containing the declaration of the accessed field. For super method invocations, the annotation contains the compile-time type of this, which determines the class that contains the declaration of the method to be invoked.

The complete typing rules are shown in Figure 5. A program is well-typed if its class definitions and final expression are well-typed. A definition, in turn, is well-typed when its field and method declarations use legal types and the method body expressions are well-typed. Finally, expressions are typed and elaborated in the context of an environment that binds free variables to types. For example, the `get` and `set` rules for fields first determine the type of the instance expression, and then calculate a class-typed field name using `ekP`; this yields both the type of the field and the class for the installed annotation. In the `set` rule, the right-hand side of the assignment must match the type of the field, but this match may exploit substitution to coerce the type of the value to a supertype. The other expression typing rules are similarly intuitive.

2.3 CLASSICJAVA Evaluation

The operational semantics for CLASSICJAVA is defined as a contextual rewriting system on pairs of expressions and stores. A store `S` is a mapping from `objects` to `class-typed` field records. A field record is a mapping from elaborated field names to values. The evaluation rules are a straightforward modification of those for imperative Scheme [13].

The complete evaluation rules are in Figure 6. For example, the `call` rule invokes a method by rewriting the method call expression to the body of the invoked method, syntactically replacing argument variables in this expression with the supplied argument values. The dynamic aspect of method calls is implemented by selecting the method based on the run-time type of the object (in the store). In contrast, the super reduction performs super method selection using the class annotation that is statically determined by the type-checker.

2.4 CLASSICJAVA Soundness

For a program of type `t`, the evaluation rules for CLASSICJAVA produce either a value that has a subtype of `t`, or one of two errors. Put differently, an evaluation cannot get stuck. This property can be formulated as a type soundness theorem.
<table>
<thead>
<tr>
<th>CLASSICALNCL(P)</th>
<th>Each class name is declared only once</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀c,c′ class c . . . class c′ . . . is in P ⇔ c \neq c′</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FIELDONCEPERCLASS(P)</th>
<th>Field names in each class declaration are unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀f,f′ class f . . . class f′ . . . is in P ⇒ f \neq f′</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>METHODONCEPERCLASS(P)</th>
<th>Method names in each class declaration are unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall m,md \cdot \text{class} \cdot \ldots \cdot \text{md} (\ldots) \cdot \text{md} (\ldots) \cdot \ldots \text{is in P} ⇒ md \neq md'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERFACESNCL(P)</th>
<th>Each interface name is declared only once</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀i,i′ interface i . . . interface i′ . . . is in P ⇒ i \neq i′</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERFACESABSTRACT(P)</th>
<th>Method declarations in an interface are abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>\exists e \cdot \text{interface} \cdot \ldots \cdot \text{md} (\ldots) \cdot {e} \cdot \ldots \text{is in P} ⇒ e is abstract</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Class is declared as an immediate subclass}</th>
</tr>
</thead>
<tbody>
<tr>
<td>c \triangleleft c′ \Rightarrow \text{class c extends c′ . . . (\ldots)} \text{is in P}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Field is declared in a class}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{c,f,d,t} \in \text{class c . . . (\ldots) . . . t . . . f . . . d . . . (\ldots)} \text{is in P}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Interface is declared as an immediate subinterface}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{i,i′} \triangleleft interface i . . . interface i′ . . . . . . \ldots \text{is in P}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Method is declared in an interface}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{md, (t₁ . . . tₙ → t), (Var₁, . . . Varₙ), c} \in \text{interface i . . . interface i′ . . . . . . . . . . . t . . . t . . . t . . . s . . . (\ldots)} \text{is in P}</td>
</tr>
</tbody>
</table>

\text{Class declares implementation of an interface} |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c \triangleleft i \Rightarrow \text{class c implements . . . i . . . . . . . . . (\ldots)} \text{is in P}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Class is a subclass}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Classes that are extended are defined}</td>
</tr>
<tr>
<td>\text{mg(\text{class c})} \subseteq \text{mg(\text{class c′})} \cup {\text{Object}}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Class hierarchy is an order}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{class c}) is antisymmetric</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Method overloading preserves the type}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall c,c′,d ∈ \text{md},T,T′,V,V′ \cdot \langle md, T, V, e \rangle \in \text{class c} \land \langle md, T′, V′, e′ \rangle \in \text{class c′} \Rightarrow (T = T′ \lor c \triangleleft c′)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Field is contained in a class}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle c′,d,t \rangle \in \text{class c′} \land c′ \in \text{class c} \land \text{class c contains} \langle c′,d,t \rangle \in \text{class c′}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Method is contained in a class}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle md, T, V, e \rangle \in \text{class c} \land c \in \text{class c′} \land \text{class c contains} \langle md, T, V′, e′ \rangle \in \text{class c′}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method is implemented in an interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall i \in \text{interface i} \cdot \exists i′ \cdot i \triangleleft i′ \cdot s.t. \cdot i \triangleleft i′ \cdot s.t. \cdot i \triangleleft i′</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface hierarchy is an order</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{interface i}) is antisymmetric</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method is implemented in an interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall i \in \text{interface i} \cdot \exists i′ \cdot i \triangleleft i′ \cdot s.t. \cdot i \triangleleft i′</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method is contained in an interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle md, T, V, e \rangle \in \text{abstract} \land \langle md, T′, V′, e′ \rangle \in \text{abstract} \Rightarrow (T = T′ \lor c \triangleleft c′)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method is implemented in an interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle md, T, V, e \rangle \in \text{abstract} \land \langle md, T′, V′, e′ \rangle \in \text{abstract} \Rightarrow (T = T′ \lor c \triangleleft c′)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods supply methods to implement interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall e,c \cdot i \Rightarrow \langle \text{md},T,V \rangle \in \text{abstract} \land \text{class c has no abstract methods (can be instantiated)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class has no abstract methods (can be instantiated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall md,T,V_{i} \cdot (md, T, V_{i}) \in \text{class c} \Rightarrow c \notin \text{abstract}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Field or method is in a type}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{class c contains} \langle \text{null}, S \rangle \land \text{class c contains} \langle \text{null}, S \rangle</td>
</tr>
</tbody>
</table>

The sets of names for variables, classes, interfaces, fields, and methods are assumed to be mutually distinct. The meta-variable T is used for method signatures of the form (t₁ . . . tₙ → t), V is used for variable lists of the form (Var₁, . . . Varₙ), and \( i \) is used for environments mapping variables to types. Ellipses on the baseline (.) indicate a repeated pattern or continued sequence, while centered ellipses (⋯) indicate arbitrary missing program text (without straddling a class or interface definition).

Figure 4: Predicates and relations in the model of CLASSICJAVA

**THEOREM:** If \( P, P' \Rightarrow P'' \cdot t \) and

- \( P \vdash (c, 0) \Rightarrow \langle \text{object}, S \rangle \) \land S(\text{object}) = \langle t', F \rangle \) and \( t' \leq P; t \)
- \( P' \vdash (c, 0) \Rightarrow \langle \text{null}, S \rangle \)
- \( P' \vdash (c, 0) \Rightarrow \langle \text{error: bad cast}, S \rangle \)
- \( P' \vdash (c, 0) \Rightarrow \langle \text{error: dereferenced null, S} \rangle \)

The main lemma in support of this theorem states that each step taken in the evaluation preserves the type correctness of the expression-store pair (relative to the program) [29]. Specifically, for a configuration on the left-hand side of an evaluation step, there exists a type environment that establishes the expression's type as some t. This environment must be consistent with the store:

\[
\begin{align*}
P, \Gamma & \vdash S \\
\Leftarrow & \langle \text{object}, c \rangle \\
= & \langle \text{null}, S \rangle \\
= & \langle \text{error: bad cast}, S \rangle \\
= & \langle \text{error: dereferenced null, S} \rangle \
\end{align*}
\]

and dom(\( F \)) ∈ dom(S) \cup \{\text{null}\}

and (\( F(c_1, c_2) \) is object and (\( c_1, c_2, F \) \( \in \text{class c} \))

and object \( \in \text{dom}(\text{class c}) \)

and dom(\( F \)) \subseteq dom(\text{class c}) \)

and object \( \in \text{dom}(\text{class c}) \)

and dom(\( F \)) \subseteq dom(\text{class c}) \)

Figure 4: Predicates and relations in the model of CLASSICJAVA

4
An evaluation step yields one of two possible configurations: either a well-defined error state or a new expression-store pair. In the latter case, there exists a new type environment that is consistent with the new store, and it establishes that the new expression has a type below $t$. A complete proof is available in an extended version of the paper [16].

2.5 Related Work on Classes

Our model for class-based object-oriented languages is similar to two recently published semantics for Java [9, 28], but entirely motivated by prior work on Scheme and ML models [13, 19, 29]. The approach is fundamentally different from most of the previous work on the semantics of objects. Much of that work has focused on interpreting object systems and the underlying mechanisms via record extensions of lambda calculi [11, 20, 24, 23, 25] or as “native” object calculi (with a record flavor) [1, 2, 3]. In our semantics, types are simply the names of entities declared in the program; the collection of types forms a DAG, which is specified by the programmer. The collection of types is static during evaluation and is only used for field and method lookups and casts. The evaluation rules describe how to transform statements, formed over the given type context, into plain values. The rules work on plain program text such that each intermediate stage of the evaluation is a complete program. In short, the model is as simple and intuitive as that of first-order functional programming enriched with a language for expressing hierarchical relationships among data types.

Dynamic class loading could be expressed in this framework as an addition to the static context. Still, the context remains the same for most of the evaluation.
3 From Classes to Mixins: An Example

Implementing a maze adventure game [17, page 81] illustrates the need for adding mixins to a class-based language. A player in the adventure game wanders through rooms and doors in a virtual world. All locations in the virtual world share some common behavior, but also differ in a wide variety of properties that make the game interesting. For example, there are many kinds of doors, including locked doors, magic doors, doors of varying heights, and doors that combine several varieties into one. The natural class-based approach for implementing different kinds of doors is to implement each variation with a new subclass of a basic door class, Door. The left side of Figure 7 shows the Java definition for two simple Door subclasses, LockedDoor and ShortDoor. An instance of LockedDoor requires a key to open the door, while an instance of ShortDoor requires the player to duck before walking through the door.

A subclassing approach to the implementation of doors seems natural at first because the programmer declares only what is different in a particular door variation as compared to some other door variation. Unfortunately, since the superclass of each variation is fixed, door variations cannot be composed into more complex, and thus more interesting, variations. For example, the LockedDoor and ShortDoor classes cannot be combined to create a new LockedShortDoor class for doors that are both locked and short.

A mixin approach solves this problem. Using mixins, the programmer declares how a particular door variation differs from an arbitrary door variation. This creates a function from door classes to door classes, using an interface as the input type. Each basic door variation is defined as a separate mixin. These mixins are then functionally composed to create many different kinds of doors.

A programmer implements mixins in exactly the same way as a derived class, except that the programmer cannot rely on the implementation of the mixin's superclass, only on its interface. We consider this an advantage of mixins because it enforces the maxim "program to an interface, not an implementation" [17, page 11].

The right side of Figure 7 shows how to define mixins for locked and short doors. The mixin LockedDoor is nearly identical to the original LockedDoor class definition, except that the superclass is specified via the interface Door. The new LockedDoor and ShortDoor classes are created by applying LockedDoor and ShortDoor to the class Door, respectively. Similarly, applying LockedDoor to ShortDoor yields a class for locked, short doors.

Consider another door variation: MagicDoor, which is similar to a LockedDoor, except the player needs a book of spells instead of a key. We can extract the common parts of the implementation of MagicDoor and LockedDoor into a new mixin, SecureDoor. Then, key- or book-specific information is composed with SecureDoor to produce LockedDoor and MagicDoor, as shown in Figure 8. Each of the new mixins extends Door since the right hand mixin in the composition, SecureDoor, extends Door.

The new LockedDoor and MagicDoor mixins can also be composed to form LockedMagicDoor. This mixin has the expected behavior: to open an instance of LockedMagicDoor, the player must have both the key and the book of spells. This combinational effect is achieved by a chain of super.canOpen() calls that use distinct, non-interfering versions of neededItem. The neededItem declarations of LockedDoor and MagicDoor do not interfere with each other because the interface extended by LockedDoor is Door, which does not contain neededItem. In contrast, Door does contain canOpen, so the canOpen method in LockedDoor overrides and chained to the canOpen in MagicDoor.

4 Mixins for Java

MIXEDJAVA is an extension of CLASSJAVA with mixins. In CLASSJAVA, a class is assembled as a chain of class expressions. Specifically, the content of a class is defined by its immediate field and method declarations and by the
class LockedDoor extends Door {
    boolean canOpen(Person p) {
        if (p.hasItem(theKey)) {
            System.out.println("You don't have the Key");
            return false;
        }
        System.out.println("Using key...");
        return super.canOpen(p);
    }
}

class ShortDoor extends Door {
    boolean canPass(Person p) {
        if (p.height() > 1) {
            System.out.println("You are too tall");
            return false;
        }
        System.out.println("Ducking into door...");
        return super.canPass(p);
    }
}

/* These classes cannot implement LockedDoor */

interface Door {
    boolean canOpen(Person p);
    boolean canPass(Person p);
}

mixin Locked extends Door {
    boolean canOpen(Person p) {
        if (p.hasItem(theKey)) {
            System.out.println("You don't have the Key");
            return false;
        }
        System.out.println("Using key...");
        return super.canOpen(p);
    }
}

class LockedDoor = Locked(Door);

mixin Short extends Door {
    boolean canPass(Person p) {
        if (p.height() > 1) {
            System.out.println("You are too tall");
            return false;
        }
        System.out.println("Ducking into door...");
        return super.canPass(p);
    }
}

class ShortDoor = Short(Door);

class LockedShortDoor = Locked(Short(Door));

Figure 7: Some class definitions and their translation to composable mixins

interface SecureDoor {
    Object neededItem();
}

mixin Secure extends Door implements SecureDoor {
    Object neededItem() { return null; }
    boolean canOpen(Person p) {
        Object item = neededItem();
        if (p.hasItem(item)) {
            System.out.println("You don't have the " + item);
            return false;
        }
        System.out.println("Using " + item + "...");
        return super.canOpen(p);
    }
}

mixin LockedNeeds extends SecureDoor {
    Object neededItem() { return theKey; }
}

mixin MagicNeeds extends SecureDoor {
    Object neededItem() { return theSpellBook; }
}

mixin Locked = LockedNeeds compose Secure;

mixin Magic = MagicNeeds compose Secure;

mixin LockedMagic = Locked compose Magic;

mixin LockedMagicDoor = LockedMagic compose Door;

class LockedDoor = Locked(Door); ...

Figure 8: Composing mixins for localized parameterization

declarations of its superclasses, up to Object. In MIXED-JAVA, a "class" is assembled by composing a chain of mixins. The content of the class is defined by the field and method declarations in the entire chain.

MIXED-JAVA provides two kinds of mixins:

- An atomic mixin declaration is similar to a class declaration. An atomic mixin declares a set of fields and methods that are extensions to some inherited set of fields and methods. In contrast to a class, an atomic mixin specifies its inheritance with an inheritance interface, not a static connection to an existing class. By abuse of terminology, we say that a mixin extends its inheritance interface.

A mixin's inheritance interface determines how method declarations within the mixin are combined with inherited methods. If a mixin declares a method x that is not contained in its inheritance interface, then that declaration never overrides another x.

An atomic mixin implements one or more interfaces as specified in the mixin's definition. In addition, a mixin always implements its inheritance interface.

- A composite mixin does not declare any new fields or methods. Instead, it composes two existing mixins to create a new mixin. The new composite mixin has all of the fields and methods of its two constituent mixins. Method declarations in the left-hand mixin override declarations in the right-hand mixin according to the left-hand mixin's inheritance interface. Composition is allowed only when the right-hand mixin implements the left-hand mixin's inheritance interface.

A composite mixin extends the inheritance interface of its right-hand constituent, and it implements all of

4We use boldfaced class to refer to the content of a single class expression, as opposed to an actual class.
Figure 9: The LockedMagicDoor\textsuperscript{m} mixin corresponds to a sequence of atomic mixins

4.1 MIXEDJAVA Programs

Figure 10 contains the syntax for MIXEDJAVA; the missing productions are inherited from the grammar of CLASSICJAVA in Figure 3. The primary change to the syntax is the replacement of class declarations with mixin declarations. Another change is in the annotations added by type elaboration. First, \textit{View} expressions are annotated with the source type of the expression. Second, a type is no longer included in the super annotation. Type elaboration also inserts extra \textit{View} expressions into a program to implement subsumption.

The predicates and relations in Figure 11 (along with the interface-specific parts of Figure 4) summarize the syntactic

Our composition operator is associative semantically, but not type-theoretically. The type system could be strengthened to make composition associative giving MIXEDJAVA a categorical flavor—by letting each mixin declare a set of interfaces for inheritance, rather than a single interface. Each required interface must then either be satisfied or propagated by a composition. We have not encountered a practical use for the extended type system.

content of a MIXEDJAVA program. A well-formed program induces a subtype relation $\leq_{p}$ on its mixins such that a composite mixin is a subtype of each of its constituent mixins.

Since each composite mixin has two supertypes, the type graph for mixins is a DAG, rather than a tree as for classes. This DAG can lead to ambiguities if subsumption is based on supertypes. For example, LockedMagic\textsuperscript{m} is a subtype of Secure\textsuperscript{m}, but it contains two copies of Secure\textsuperscript{m} (see Figure 9), so an instance of LockedMagic\textsuperscript{m} is ambiguous as an instance of Secure\textsuperscript{m}. More concretely, the fragment

```
LockedMagicDoor\textsuperscript{m} door = new LockedMagicDoor\textsuperscript{m};
(view Secure\textsuperscript{m} door).neededItem();
```

is ill-formed because LockedMagic\textsuperscript{m} is not viewable as Secure\textsuperscript{m}. The "viewable as" relation $\leq_{p}$ is a restriction on the subtype relation that eliminates ambiguities. Subsumption is thus based on $\leq_{p}$ rather than $\leq_{s}$. The relations $\leq_{p}$, which collect the fields and methods contained in each mixin, similarly eliminate ambiguities.

4.2 MIXEDJAVA Type Elaboration

Despite replacing the subtype relation with the "viewable as" relation for subsumption, CLASSICJAVA's type elaboration strategy applies equally well to MIXEDJAVA. The typing rules in Figure 12 are combined with the defin', meth, let, var, null, and abs rules from Figure 5.

Three of the new rules deserve special attention. First, the super\textsuperscript{m} rule allows a super call only when the method is declared in the current mixin's inheritance interface, where the current mixin is determined by looking at the type of this. Second, the weak\textsuperscript{m} rule strips out the view part of the expression and delegates all work to the subsumption rules. Third, the sub\textsuperscript{m} rule for subsumption inserts a \textit{View} operator to make subsumption coercions explicit.

4.3 MIXEDJAVA Evaluation

The operational semantics for MIXEDJAVA differs substantially from that of CLASSICJAVA. The rewriting semantics of
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mix</td>
<td>Each mixin name is declared only once. ( \forall m, m' ) ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( P ) if and only if ( m \neq m' )</td>
</tr>
<tr>
<td>field</td>
<td>Field names in each mixin declaration are unique. ( \forall f, f' ) ( \text{mix} ) ( f \ldots \text{mix} ) ( f' \ldots ) is in ( P ) if and only if ( f \neq f' )</td>
</tr>
<tr>
<td>method</td>
<td>Method names in each mixin declaration are unique. ( \forall m, m' ) ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( P ) if and only if ( m \neq m' )</td>
</tr>
<tr>
<td>noAbstract</td>
<td>Methods in a mixin are not abstract. ( \forall m \notin \text{Abstract} ) ( m \ldots \text{abstract} ) ( m' \ldots ) is in ( P ) if and only if ( m \neq m' )</td>
</tr>
<tr>
<td>.mixin</td>
<td>Mixin declares an inheritance interface. ( \forall i ) ( \text{mix} ) ( i \ldots \text{extends} ) ( \ldots ) ( \text{mix} ) ( e \ldots ) is in ( P ) if and only if ( \text{mix} ) and ( \text{mix} )</td>
</tr>
<tr>
<td>includes</td>
<td>Mixin declares an inheritance interface. ( \forall i ) ( \text{mix} ) ( i \ldots \text{extends} ) ( \ldots ) ( \text{mix} ) ( e \ldots ) is in ( P ) if and only if ( \text{mix} ) and ( \text{mix} )</td>
</tr>
<tr>
<td>inherits</td>
<td>Mixin is a submixin of ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( \text{mix} ) if and only if ( m \neq m' ) and ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( \text{mix} )</td>
</tr>
<tr>
<td>view</td>
<td>Mixin can be viewed as a mixin (i.e., ( \text{mix} ) is a submixin of ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( \text{mix} ) if and only if ( m \neq m' ) and ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( \text{mix} ))</td>
</tr>
<tr>
<td>complete</td>
<td>Mixins that are composed are defined. ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( P ) if and only if ( m \neq m' ) and ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( \text{mix} )</td>
</tr>
<tr>
<td>with</td>
<td>Mixin hierarchy is an order. ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( P ) if and only if ( m \neq m' ) and ( \text{mix} ) ( m \ldots \text{mix} ) ( m' \ldots ) is in ( \text{mix} )</td>
</tr>
<tr>
<td>completeInterfaces</td>
<td>Extended/implemented interfaces are defined. ( \forall i ) ( \text{mix} ) ( i \ldots \text{extends} ) ( \ldots ) ( \text{mix} ) ( e \ldots ) is in ( P ) if and only if ( \text{mix} ) and ( \text{mix} )</td>
</tr>
<tr>
<td>extends</td>
<td>Mixin extends an interface. ( \forall i ) ( \text{mix} ) ( i \ldots \text{extends} ) ( \ldots ) ( \text{mix} ) ( i \ldots ) is in ( P ) if and only if ( \text{mix} ) and ( \text{mix} )</td>
</tr>
<tr>
<td>implements</td>
<td>Mixin implements an interface. ( \forall i ) ( \text{mix} ) ( i \ldots \text{implements} ) ( \ldots ) ( \text{mix} ) ( i \ldots ) is in ( P ) if and only if ( \text{mix} ) and ( \text{mix} )</td>
</tr>
</tbody>
</table>

Figure 11: Predicates and relations in the model of MIXEDJAVA.
explosion in the size of the program, which occurs when \( m_1 \) and \( m_2 \) are actually the same mixin \( m \). Since the mixin \( m \) represents a type, renaming \( x \) in each use of \( m \) splits it into two different types, which requires type-splitting at every expression in the program involving \( m \).

Our MIXED.JAVA semantics handles the duplication of method names with run-time context information: the current \textit{view} of an object.\(^6\) During evaluation, each reference to an object is bundled with its view of the object, so that values are of the form \((\text{object} || \text{view})\). A reference's view can be changed by subsumption, method calls, or explicit casts. A view is represented as a chain of mixins. This chain is always a tail of the object's full chain of mixins, i.e., the chain of mixins for the object's instantiation type. The tail designates a specific point in the full mixin chain for selecting methods during dynamic dispatch. For example, when an instance of LockedMagicDoor\(^m\) is used as a Magic\(^m\) instance, the view of the object is [MagicNeeded\(^m\) Secure\(^m\) Door\(^m\)]. With this view, a search for the neededItem method of the object begins in the MagicNeeded\(^m\) element of the chain.

The first phase of a search for some method \( x \) locates the \textit{base declaration} of \( x \), which is the unique non-overriding declaration of \( x \) that is visible in the current view. This declaration is found by traversing the view from left to right, using the inheritance interface at each step as a guide for the next step (via the \( \alpha \) and \( \tau \) relations). When the search reaches a mixin whose inheritance interface does not include \( x \), the base declaration of \( x \) has been found. But the base declaration is not the destination of the dispatch; the destination is an overriding declaration of \( x \) for this base that is contained in the object's instantiated mixin. Among the declarations that override this base, the leftmost declaration is selected as the destination. The location of that overriding declaration determines both the method definition that is invoked and the view of the object (i.e., the representation of \textit{this}) within the destination method body. This dispatching algorithm is encoded in the \( \tau \) relation.

Let us apply the algorithm to \texttt{a.gte()}\(^m\) in the following example:

\begin{verbatim}
Mixin Getter\(^m\) extends Empty {
    Object ge(Secure\(^m\) o) { o.neededItem() }
}

let door = new LockedMagicDoor\(^m\)
let g = new Getter\(^m\)
in g.gte(view Secure\(^m\) view Locked\(^m\) door);  
g.gte(view Secure\(^m\) view Magic\(^m\) door)
\end{verbatim}

For the first call to \texttt{a.gte()}, \( o \) is replaced by a reference with the view \[Secure\(^m\) MagicNeeded\(^m\) Secure\(^m\) Door\(^m\)]\(^m\). In this view, the base declaration of \textit{neededItem} is in the leftmost Secure\(^m\) since \textit{neededItem} is not in the interface extended by Secure\(^m\). The overriding declaration is in LockedNeeded\(^m\), which appears to the left of Secure\(^m\) in the instantiated chain and extends an interface that contains \textit{neededItem}.
$$E = \left\{ \begin{array}{ll} E : m : j d & \text{if } E : m : j d \in E \\
\text{error(}\ldots\text{)} & \text{if } \text{null}(v) \ldots e \ldots \\
\text{super} \equiv \equiv (\ldots \text{null}(v) \ldots e \ldots ) & \text{in } E \\
\text{view}(\ldots E \ldots ) & \text{let } \text{var} = E \text{ in } e \\
\end{array} \right.$$

$$P \downarrow E \left\{ \begin{array}{ll} (\text{new } m, S) \leftarrow (\text{object})(M), S \leftarrow (\text{object} \rightarrow (m, M, J d = \text{null}, \ldots, m_{n} = J d \rightarrow \text{null})) & \text{[new]} \\
m : J d \in \text{dom}(S) \text{ and } m \rightarrow P M & \text{[def]} \\
S(\text{object}) \equiv (m, f) \text{ and } M_{t} \rightarrow M' \text{ and } F(M' J d) = v & \text{[def]} \\
S(\text{object}) \equiv (m, f) \text{ and } M_{t} \rightarrow M' & \text{[call]} \\
S(\text{object}) \equiv (m, f) \text{ and } m \rightarrow P M_{o} \text{ and } m_{n} = J d \rightarrow M' \text{ in } M_{o} & \text{[super]} \\
m \rightarrow P t \text{ and } M_{t} \rightarrow M_{t} \text{ and } m \rightarrow P M_{t} \text{ and } M_{t} / t \rightarrow M_{t} & \text{[view]} \\
S \leftarrow (\text{object})(M') \text{ in } M' & \text{[cast]} \\
S \leftarrow (\text{object})(M') \text{ in } M' & \text{[let]} \\
S \leftarrow (\text{error: bad cast, } S) & \text{[error]} \\
S \leftarrow (\text{error: dereferenced null, } S) & \text{[null]} \\
S \leftarrow (\text{error: dereferenced null, } S) & \text{[null]} \\
\end{array} \right.$$ 

Figure 13: Operational semantics for MIXEDJAVA

In contrast, the second call to $g$.get receives a reference with the view [Secure$^{\text{M}}$ Door$^{\text{M}}$]. In this view, the base definition of neededItem is in the rightmost Secure$^{\text{M}}$ of the full chain, and it is overridden in MagicNeeded$^{\text{M}}$. Neither the definition of neededItem in LockedNeeded$^{\text{M}}$ nor the one in the leftmost occurrence of Secure$^{\text{M}}$ is a candidate relative to the given view, because Secure$^{\text{M}}$ extends an interface that hides neededItem.

MIXEDJAVA not only differs from CLASSICJAVA with respect to method dispatching, but also in its treatment of super. In MIXEDJAVA, super dispatches are dynamic, since the "super" method for a super expression is not statically known. The super dispatch for mixins is implemented like regular dispatches with the $\epsilon$ dispatch rule, but using a tail of the current view in place of both the instantiation and view chain; this ensures that a method is selected from the leftmost mixin that follows the current view.

Figure 13 contains the complete operational semantics for MIXEDJAVA as a rewriting system on expression-store pairs, like the class semantics described in Section 2.3. In this semantics, an object in the store is tagged with a mixin instead of a class, and the values are null and (object)[] pairs.

4.4 MIXEDJAVA Soundness

The type soundness theorem for MIXEDJAVA is mutatis mutandis the same as the soundness theorem for CLASSICJAVA as described in Section 2.4. To prove the soundness theorem, we introduce a conservative extension, MIXEDJAVA$'$, which is defined by revising some of the MIXEDJAVA relations (see Figure 14).

In the extended language, the subtype relation is used directly for the "viewable as" relation without eliminating ambiguities. Thus, MIXEDJAVA$'$ allows coercions and method calls that are rejected as ambiguous in MIXEDJAVA. This makes MIXEDJAVA$'$ less suitable as a programming language, but the proof of its type soundness theorem is significantly simpler. The soundness theorem for MIXEDJAVA$'$ then applies to MIXEDJAVA by the following two lemmas:

1. Every MIXEDJAVA program is a MIXEDJAVA$'$ program.
2. $P \downarrow \langle e, S \rangle \leftarrow \langle e', S' \rangle$ in MIXEDJAVA
   $\Rightarrow P \downarrow \langle e, S \rangle \leftarrow \langle e', S' \rangle$ in MIXEDJAVA$'$.

A complete definition of MIXEDJAVA$'$, its soundness proof, and proofs for the above lemmas are available in the extended paper [16].

4.5 Implementation Considerations

The MIXEDJAVA semantics is formulated at a high level, leaving open the question of how to implement mixins efficiently. Common techniques for implementing classes can be applied to mixins, but two properties of mixins require new implementation strategies. First, each object reference must
carry a view of the object. This can be implemented using double-wide references, one half for the object pointer and the other half for the current view. Second, method invocation depends on the current view as well as the instantiation mixin of an object, as reflected in the $\mathcal{E}_\mathcal{B}$ relation. Nevertheless, this relation determines a static, per-mixin method table that is analogous to the virtual method tables typically generated for classes.

The overall cost of using mixins instead of classes is equivalent to the cost of using interface-typed references instead of class-typed references. The justification for this cost is that mixins are used to implement parts of a program that cannot be easily expressed using classes. In a language that provides both classes and mixins, portions of the program that do not use mixins do not incur any extra overhead.

4.6 Related Work on Mixins

Mixins first appeared as a CLOS programming pattern [21, 22]. Unfortunately, the original linearization algorithm for CLOS's multiple inheritance breaks the encapsulation of class definitions [10], which makes it difficult to use CLOS for proper mixin programming. The CommonLisp [27] dialect of CLOS supports multiple inheritance without breaking encapsulation, but the language does not provide simple composition operators for mixins.

Bracha has investigated the use of "mixin modules" as a general language for expressing inheritance and overriding in objects [5, 6, 7]. His system is based on earlier work by Cook [8]; its underlying semantics was recently reformulated in categorical terms by Ancona and Zucca [4]. Bracha's system gives the programmer a mechanism for defining modules (classes, in our sense) as a collection of attributes (methods). Modules can be combined into new modules through various merging operators. Roughly speaking, these operators provide an assembly language for expressing class-to-class functions and, as such, permit programmers to construct mixins. However, this language forces the programmer to resolve attribute name conflicts manually and to specify attribute overriding explicitly at a mixin merge site. As a result, the programmer is faced with the same problem as in Common Lisp, i.e., the low-level management of details. In contrast, our system provides a language to specify both the content of a mixin and its interaction with other mixins for mixin compositions. The latter gives each mixin an explicit role in the construction of programs so that only sensible mixin compositions are allowed. It distinguishes method overriding from accidental name collisions and thus permits the system to resolve name collisions automatically in a natural manner.

5 Conclusion

We have presented a programming language of mixins that relies on the same intution as single inheritance classes. Indeed, a mixin declaration in our language hardly differs from a class declaration since, from the programmer's local perspective, there is little difference between knowing the properties of a superclass as described by an interface and knowing the exact implementation of a superclass. However, from the programmer's global perspective, mixins free each collection of fields and method extensions from the tyranny of a single superclass, enabling new abstractions and increasing the re-use potential of code.

While using mixins is inherently more expensive than using classes because mixins enforce the distinction between implementation inheritance and subtyping—the cost is reasonable and offset by gains in code re-use. Future work on mixins must focus on exploring compilation strategies that lower the cost of mixins, and on studying how designers can exploit mixins to construct better design patterns.

Acknowledgements. Thanks to Corley Cartwright, Robby Findler, Cormac Flanagan, and Dan Friedman for their comments on early drafts of this paper.

References


