Selective Method Combination in Mixin-Based Composition

Tetsuo Kamina Tetsuo Tamai
The University of Tokyo
3-8-1, Komaba, Meguro-ku, Tokyo,
153-8902, Japan
{kamina,tamai}@graco.c.u-tokyo.ac.jp

ABSTRACT

A mixin is a reusable module that provides uniform extensions and modifications to classes. It is an abstract subclass that is composable with a variety of superclasses. In mixin-based composition, however, the problem of *accidental overriding* arises. A method declared in a mixin may *accidentally* overrides its superclasses' method. To tackle this problem, we propose a new approach of method lookup that allows *selective method combination*; that is, when we have multiple methods with the same name and the same formal parameter types in a composition, we can select which method to execute, and which method is called when there exists a method call to `super`. This proposal is an extension of hygienic mixins with stronger expressive power. This proposal is implemented in McJava, an extension of Java with mixin-types. Its compilation is achieved by source code translation to Java thus making it runnable on a standard Java virtual machine.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*inheritance*

General Terms

Language design and implementation

Keywords

McJava, Accidental overriding, Hygienic mixins, Mixin-based subtyping

1. INTRODUCTION

A *mixin* [3] is a reusable module that provides uniform extensions and modifications to classes. It is a partially implemented subclass that is composable with a variety of "superclasses." Compared with single inheritance scheme, mixin-based composition provides much reusability because it has an ability to add common features (that will be duplicated in a single inheritance hierarchy) to a variety of classes. Mixin-based composition has been popularized by CLOS [13] and currently integrated with main-stream strongly typed object-oriented languages [2, 8].

One problem of mixin-based composition is known as *accidental overriding* [1]. While a subclass in many object-oriented languages explicitly declares its superclass, in mixin-based composition, a mixin does not know which superclass the mixin will be composed with. Therefore, when a user of a mixin (who will be different from an implementor of that mixin) tries to compose it with some other classes, it is possible that a method declared in the mixin accidentally overrides a method declared in the superclass.

In general, there are two kinds of overriding: *intentional* overriding and *accidental* overriding. In the case of intentional overriding, we know that a superclass has a method that will be overridden. In this case, we explicitly declare methods imported from the superclass (e.g. as explained in the following sections, we can use `requires` clause for this purpose in programming language McJava [8]), then override them in a mixin. In the case of accidental overriding, on the other hand, we do not know that the superclass has a method whose name and formal parameter types are the same as those of a method declared in the mixin. This overriding is harmful because it accidentally changes the behavior [10] of the superclass.

One way to avoid accidental overriding is to have a compiler reject a program that contains a composition with accidental overriding. Of course, we can statically analyze whether there is accidental overriding or not. However, this approach limits the reusability of mixins. To promote reusability of mixins, mixins should be composed with classes even when there exists accidental overriding. Another way to avoid accidental overriding is to select which method to be invoked by using the context information that encloses the method invocation. Furthermore, we should also consider that, in Java-like languages, we may *combine* the overriding method with the overridden (the original) method by calling the latter method with `super`. If we allow the selective method invocation as mentioned above, there may exist multiple candidates for *combination* of methods[1]. We need a new mechanism of method lookup.

In this paper, we propose a new approach to method lookup that solves the accidental overriding problem. Our

[1]The source of the term "method combination" is CLOS [9].

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

```
class Person {
  String _name;
  String name() { return _name; }
}
mixin Employee requires { String name(); } {
  String id, title;
  String name() { return title+super.name(); }
  String getID() { return id; }
}
mixin Student {
  String id;
  String getID() { return id; }
}
class Main {
  public static void main(String[] args) {
    Employee e =
      new Student::Employee::Person();
    String id = e.getID();
    ...
  }
}
```

Figure 1: Accidental Overriding in McJava

approach allows *selective method combination*; that is, if we have multiple candidates for method call to `super`, we can select which method to be called. This selection is achieved by using static type information that is equipped with a flexible version of inheritance-based subtyping. We have implemented this mechanism into McJava, an extension of Java with mixin-types proposed by Kamina and Tamai [8]. Our implementation technique is an extension of *hygienic mixins* [5, 1, 12]. Since McJava provides very flexible subtyping rules, applying the implementation techniques of hygienic mixins is actually a non-trivial issue. In McJava, an immediate superclass of a mixin in the run-time inheritance chain may be different from the statically known superclass thus requiring more sophisticated treatment in invoking a superclass's method.

Our approach may look specific only to be applied to McJava because it depends on McJava subtyping rules. However, some languages such as `gbeta` [4] allow similar mechanism as McJava. We believe that the proposal of this paper can be applicable to such languages. Furthermore, as shown in the following sections, subtyping in McJava is a generalization of inheritance-based subtyping. When this subtyping scheme is introduced into other languages, the problem treated in this paper always arises and the proposed solution may be useful.

The rest of this paper is structured as follows. In section 2, we show the problem of accidental overriding and explain why the selective method combination is required. In section 3, we propose a new method lookup mechanism that solves the aforementioned problem. In section 4, we sketch how to implement the proposed approach in the McJava compiler that translates McJava programs to Java programs. Section 5 compares this work with other related work, and section 6 concludes.

2. THE PROBLEM OF ACCIDENTAL OVERRIDING

In Figure 1, we illustrate the problem of accidental overriding by using McJava programming language. The statement beginning with `mixin` is a *mixin declaration*. A mixin declaration has the following form:

`mixin` X [`requires` I] { ... }

where X denotes the name of mixin and I denotes the interface that the mixin *requires*. This means that classes that implement interface I can be composed with mixin X. For example, class `Person` can be composed with mixin `Employee`, because it implements the interface that the mixin `Employee` requires(i.e. `String name()` method). The imported methods declared in the `requires` clause can be referred in the body of mixin, e.g., `super.name()` called inside `Employee.name()`. In other words, `Employee` intentionally overrides the method `String name()`.

Mixin `Employee` can be composed with class `Person`, and this composition is written as `Employee::Person`. This composition is regarded as a subclass derived from the `Person` class, with subclass body declaration being the same as the body of `Employee`. In Figure 1, this composition is further composed with another mixin `Student`.

The mixin `Employee` also declares method `String getID()` that returns the identification number at the company, and the mixin `Student` declares the same method that returns the identification number at the school. In class `Main`, we compose `Student` with `Employee` and `Person` and create its instance (which means an employee who is also a student). This instance is referred by variable `e` whose static type is `Employee`. When `getID()` method is invoked on `e`, we expect `Employee.getID()` to be executed; however, if the normal method lookup rule of Java stipulating the most specific method to be always selected is applied, `Student.getID()` is called. Because it behaves differently from `Employee.getID()`, the result of method call `e.getID()` does not satisfy the expectation of the user of `e`. Therefore, in this case the alternative method lookup scheme is required.

By preserving the static type information of variable `e`, we can invoke `Employee.getID()` instead of `Student.getID()`. This mechanism is known as *hygienic mixins* [1, 12]. If we adopt this scheme, there can be more than one method that has the same name and the same formal parameter types on that composition. We may select a method to be invoked by using static type information. Furthermore, if we intentionally override the `getID()` method in a possible subclass of that composition, then there will exist multiple *combinations* of methods: methods combined by calling the original method with `super`. To show when this situation occurs, we use the following example.

Suppose we have a mixin `Id` that imports a method `String getID()` from a superclass, and intentionally override it.

```
mixin Id requires { String getID(); }{
  String getID() { return super.getID(); }
  ...
}
```

This mixin implements a concern of identification, performing identification-related tasks. The `getID()` method declared in that mixin calls `super.getID()` and returns its result. This method is regarded as an abstract method that can be called by other methods declared in that mixin. This is a variety of *template design pattern* [6].

We can compose `Id` with `Employee` and `Student`, adding identification-specific operations to those mixins. Furthermore, as shown previously, an employee may also become a student. We have the following composition:

```
Id::Student::Employee p =
  new Id::Student::Employee::Person();
processIdOfEmployee(p);
processIdOfStudent(p);
```

In this case, both of `Employee` and `Student` provides `String getID()` method. Then, a question arises; when `Id.getID()` executes the expression `super.getID()`, which method should be called, `Employee.getID()` or `Student.getID()`?

The answer to the question depends on the static typing of the instance referred by the variable `p`. Suppose the `processIdOfEmployee` method is declared as follows:

```
void processIdOfEmployee(Id::Employee e) {
  String id = e.getID();
  ...
}
```

McJava allows a composition `Id::Student::Employee` to be a subtype of `Id::Employee`, which means, in McJava, subtype relations are not restricted to the immediate inheritance relations. In `Id::Employee`, `Id` immediately inherits definitions from `Employee`. In `Id::Student::Employee`, `Id` transitively inherits definitions from `Employee`. The composition `Id::Employee` has the same members with `Id::Student ::Employee`, so the latter is conceptually a subtype of the former. Therefore, the instance of latter can safely be typecasted to the former. This subtyping is a generalization of inheritance-based subtyping that promotes reusability of programs.

In the above case, local variable `e` has type `Id::Employee`; therefore, the executed code of `super.getID()` in `Id.getID()` should be `Employee.getID()`.

On the other hand, the definition of `processIdOfStudent` is:

```
void processIdOfStudent(Id::Student e) {
  String id = s.getID();
  ...
}
```

In this case, local variable `s` has static type `Id::Student`; therefore, the executed code of `super.getID()` in `Id.getID()` should be `Student.getID()`. Therefore, in this case we should have multiple method combinations: [`Id.getID()`, `Employee.getID()`] and [`Id.getID()`, `Student.getID()`].

3. SOLVING THE PROBLEM BY USING SELECTIVE METHOD COMBINATION

To tackle the problem, we propose a new method lookup scheme that allows selective method combination i.e. when we have multiple candidates for method call to `super`, we can select which method to execute by using the static type information. To explain our approach, we assume that mixins `A`, `B`, `C`, `D` and a class `E` have a method `void m()`. Mixins `B` and `D` also require a method `void m()` and call `super.m()` inside the definition of `B.m()` and `D.m()`, which means they intentionally override a method `void m()`. Finally, an instance of a composition `A::B::C::D::E` is created and stored into a local variable `o` whose static type is `B::D` (Figure 2):

```
B::D o = new A::B::C::D::E();
o.m();
```

In this case, `A.m()` and `C.m()` accidentally override the superclass method, and `B.m()` and `D.m()` intentionally override the superclass method. Because the method `o.m()` is invoked with the static scope `B::D`, the method that `B.m()` overrides should be `D.m()`. Since `C.m()` accidentally overrides `D.m()`, the executed method should be `B.m()` and `D.m()` (followed by `E.m()`).

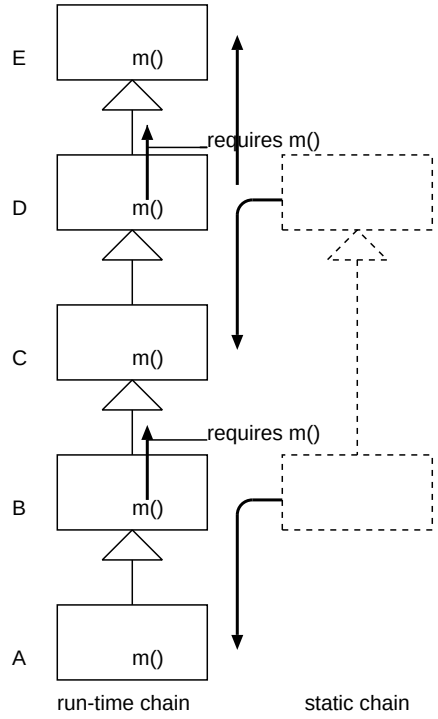

Figure 2: New method lookup in McJava

We sketch the method lookup algorithm as follows:

1. In our approach, the method lookup (e.g. `o.m()`) starts with the bottom of *static* inheritance chain (that is `B` in Figure 2. We mean a static inheritance chain by a statically known inheritance relationship to distinguish it with the *run-time* inheritance chain. The static inheritance chain is denoted with dashed lines in Figure 2), then searches *down* the *run-time* inheritance chain.

2. In each mixin definition in the run-time inheritance chain, the method lookup searches a method with the same name and the same formal parameter types as the invoked method.

   In Figure 2, it finds that `A` has a definition of `void m()`.

3. If the found method intentionally overrides the superclass's method i.e. a method with the same name and the same formal parameter types is declared in the `requires` clause, the search goes down further to follow the longest possible chain of intentional overriding. If the method is not declared in the `requires`

clause, this is an accidental overriding so the down search stops and the last matched method encountered before reaching the mixin that hides the method is executed.

In Figure 2, `A` does not require a method `void m()`; therefore, the resolved method is `B.m()`.

4. The method lookup then searches the superclass's method called on `super`. This search goes *up* on the run-time inheritance chain until it reaches the starting point (`B` in Figure 2). After reaching the starting point, the search then goes *up* the next mixin of *static inheritance chain*, and searches *down* the run-time inheritance chain again.

   In Figure 2, `super.m()` is called during the execution of `B.m()`. The method lookup then searches down the run-time inheritance chain from mixin `D`.

5. The method lookup iterates the searching process 1 through 4 until no combined methods are left.

   In Figure 2, the method lookup finds that `C` has a definition of `void m()`; however, `C` does not import a method `void m()`. Therefore, the method call `super.m()` in `B.m()` results in the execution of `D.m()`. During the execution of `D.m()`, `super.m()` is called, which results in the execution of `E.m()`.

So far, the executed methods in Figure 2 are `B.m()`, `D.m()` and `E.m()`. In other words, the method combination from `A.m()`, `B.m()`, `C.m()`, `D.m()` and `E.m()` with a static scope `B::D` is [`B.m()`, `D.m()`, `E.m()`].

Note that if the pure-Java semantics of method lookup is applied, the executed method is `A.m()`.

4. IMPLEMENTATION

We have implemented the mechanism explained above into the McJava compiler that compiles McJava source programs into Java source programs. Java virtual machine does not preserve static type information of run-time objects. To preserve static type information in translated Java programs, the compiler changes the name of methods declared in mixins and corresponding method invocations.

McJava compilation strategy is explained by Kamina and Tamai in [8]. In this paper, we briefly sketch how the renaming of methods works in the compilation. Figure 3 shows the translated Java code from the definitions in Figure 1 and `Id` in section 2:

1. All the method names newly introduced in a mixin are prefixed by the name of that mixin and a character `$`. For example, the `getID()` method in the mixin `Employee` becomes `Employee$getID()`. This renaming avoids accidental overriding.

2. The treatment of methods that intentionally override superclass's methods is more sophisticated. Firstly, not as in the case of accidental overriding, the compiler does not change the name of the method, but changes the method name of `super` call to the name of the overridden method in the *translated* class hierarchy. For example, the `super` call inside `getID()` method in mixin `Id` becomes `Student$getID()` in the translated class (`Id_Person`). If there exist multiple method combinations, the compiler also inserts new methods whose names are the same as those of overridden methods, copying body of the overriding method. For example, the method declaration `getID()` in `Id` is also copied into the method declarations `Student$getID()` and `Employee$getID()` in the translated class. Note that the name of the method in method invocation on `super` is also changed appropriately.

```
class Person {
  String _name;
  String name() { return _name; }
}
interface Employee {
  String name();
  String Employee$getID();
}
class Employee_Person extends Person
      implements Employee {
  String id, title;
  String name() { return title+super.name(); }
  String Employee$getID() { return id; }
}
interface Student {
  String Student$getID();
}
class Student_Person extends Employee_Person
      implements Student {
  String id;
  String Student$getID() { return id; }
}
interface Id {
  String getID(); ...;
}
class Id_Person extends Student_Person
      implements Id {
  String Student$getID() {
    return super.Student$getID(); }
  String Employee$getID() {
    return super.Employee$getID(); }
  String getID() {
    return super.Student$getID(); }
  ...
}
interface Id_Employee extends Employee,Id { }

class Id_Student_Employee_Person
      extends Id_Person
      implements Id_Employee {
}
```

Figure 3: Compiled code of Figure 1 and `Id`

The method name in corresponding method invocation is also changed. For example, the declaration of `processIdOfEmployee` in section 2 becomes the following declaration:

```
void processIdOfEmployee(Id_Employee e) {
  String id = e.Employee$getID();
  ...
}
```

5. RELATED WORK

As mentioned earlier, our approach is an extension of hygienic mixins [1, 12]. The implementation of hygienic mixins is based on MixedJava, formalized by Flatt et al. [5]. MixedJava uses run-time context information, called *view*, to determine which method should be invoked when an accidental overriding exists. The subtyping rules of these work do not allow an immediate superclass of a mixin in run-time inheritance chain to be different from the statically known superclass. The selective call of the "original" method to `super` is not achieved in [1, 12, 5].

Ernst proposed the *propagation* mechanism of method combination in the statically typed language `gbeta` [4], a generalization of the language BETA [11]. `gbeta` also provides similar mechanism with our approach that allows two methods with the same signature to coexists in the same object, and to select which one of them to call based on the statically known type of the receiver. However, BETA/`gbeta` does not provide Java-style method overriding; instead it provides method *argumentation* by `INNER` statements. Therefore, the result of selective method combination in `gbeta` is different from our approach. Actually there is a design tradeoff; further information about it is found in [3]. We also note that recently Goldberg et al. propose a language that integrates `super` and `INNER` [7].

Traits [14] resolve naming conflicts (i.e. accidental overriding) by aliasing of conflicting methods and making the original method invisible from outside. This solution alleviates the problem only to small extent and requires other than language features such as good refactoring tools, while our approach solves the problem purely in language design and implementation.

Epsilon [16, 15] is a role-based executable model that has a feature of dynamic object adaptation. When an Epsilon object dynamically adapt to a role, replacing of methods may occur. This replacing allows more flexible method combination than the traditional method overriding where the name of overridden method is always the same as that of overriding method. Even though McJava does not allow this replacing, we consider the mechanism proposed in this paper provides a good basis for incorporating similar mechanism into Epsilon.

6. CONCLUDING REMARKS

In this paper, we have proposed a new method lookup scheme of selective method combination. This approach solves the problem of accidental overriding in mixin-based composition. With the flexible subtyping mechanism defined in McJava, in the case of having multiple candidates for method call to `super` we can select which method to be called. This approach promotes flexibility of mixin-based compositions, and reliability of programs because our approach makes it easier to ensure the behavior of classes. Our approach can be implemented as the source code translation into Java programs thus making it runnable on a standard Java virtual machine.

Acknowledgements: The research has been conducted under Kumiki Project, supported as a Grant-in-Aid for Scientific Research (13224087) by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan.

7. REFERENCES

[1] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proceedings of OOPSLA2003*, pages 96–114, 2003.

[2] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, 2003.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

[4] Erik Ernst. Propagating class and method combination. In *ECOOP'99*, volume 1628 of *LNCS*, pages 67–91. Springer-Verlag, 1999.

[5] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL 98*, pages 171–183, 1998.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] David S. Goldberg, Robert Bruce Findler, and Matthew Flatt. Super and inner – together at last! In *OOPSLA 2004*, pages 116–129, 2004.

[8] Tetsuo Kamina and Tetsuo Tamai. McJava – a design and implementation of Java with mixin-types. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004*, volume 3302 of *LNCS*, pages 398–414. Springer-Verlag, 2004.

[9] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[10] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[11] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[12] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of OOPSLA2001*, pages 211–222, 2001.

[13] D. A. Moon. Object-oriented programming with flavors. In *OOPSLA'86 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 1–8, 1986.

[14] Nathanael Schärli, Stepḥane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP 2003*, LNCS 2743, pages 248–274, 2003.

[15] Tetsuo Tamai. Evolvable Programming based on Collaboration-Field and Role Model. In *International Workshop on Principles of Software Evolution (IWPSE'02)*, pages 1–5, 2002.

[16] Naoyasu Ubayashi and Tetsuo Tamai. Separation of Concerns in Mobile Agent Applications. In *Metalevel Architectures and Separation of Crosscutting Conserns – Proceedings of the 3rd International Conference (Reflection 2001)*, volume 2192 of *LNCS*, pages 89–109. Springer-Verlag, 2001.