JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 4, No. 10, 2005

Flexible Method Combination based on Mixin Subtyping

Tetsuo Kamina, JST CREST, Japan
Tetsuo Tamai, The University of Tokyo, Japan

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* override its superclass's method. To tackle this problem, we propose a *selective method combination* that selects where the method invocation starts, and where method body execution jumps in the case of `super` invocation, by using the static type information of the receiver. We use the flexible mixin-based subtyping rules that allow subtype relations to be not restricted to the immediate inheritance relations; therefore, flexible control of method combination is achieved. To describe precisely how selective method combination works, we formalize this mechanism over McJava, an extension of Java with mixin-types. We also implement this mechanism by source code translation to Java thus making it runnable on a standard Java virtual machine.

1 INTRODUCTION

A *mixin* [?] 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 more reusability because it has the 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 [?, ?] and there are many attempts to integrate mixins with mainstream strongly typed object-oriented languages [?, ?].

One problem of mixin-based composition is known as *accidental overriding* [?]. Since a mixin does not know which superclass the mixin will be composed with, the mixin may accidentally override methods declared in the superclass. This overriding should be avoided, because it harmfully changes the behavior of the superclass. This problem is not so simple, however, because a mixin may also *intentionally* overrides its superclass's methods (in this case, we explicitly declare methods imported from the superclass, then override them in a mixin; e.g., as explained in the following sections, we can use the `requires` clause for this purpose in programming language McJava [?]). Furthermore, we should consider that, in Java-like languages, we may *combine* the overriding method with the overridden (the original) method by using `super` invocation. Therefore, if we have an inheritance chain of mixins with mixture of intentional and accidental overriding, we should consider where the execution of method invocation should start, and where method body execution should jump when it contains `super`. We need a new mechanism of method lookup.

In this paper, we propose a *selective method combination* that selects where the method invocation starts, and which method body executes in the case of `super` invocation, by using the static type information of the receiver. In this paper, we use the flexible mixin-based subtyping rules that allow subtype relations to be not restricted to the immediate inheritance relations; therefore, flexible control of method combination is achieved. To describe precisely how the selective method combination works, we have formalized this mechanism over McJava, an extension of Java with mixin-types [?], by extending dynamic semantics of Core McJava, the core calculus of McJava. We have also implemented this mechanism. The implementation technique used in the proposed language is an extension of *hygienic mixins* [?]. Since McJava provides very flexible subtyping rules, applying the implementation techniques of hygienic mixins is actually a non-trivial issue.

Our approach may look specific only to be applied to McJava because it depends on McJava subtyping rules. However, some languages such as `gbeta` [?] allow similar mechanism with 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 formalize the proposed system. In section 5, we sketch how to implement the proposed approach in the McJava compiler that translates McJava programs to Java programs. Section 6 compares this work with other related work, and section 7 concludes this paper.

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()`

```
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

method). Note that McJava uses *structural subtyping* instead of *nominal subtyping* to determine whether such composition is allowed or not. While nominal subtyping (by using named interfaces) gives more exact information of composable superclasses, structural subtyping increases flexibility of composition. Mixins are often composed with many other classes that do not share the same named interfaces (e.g., `Person` in Figure 1 does not `implements` any interfaces); therefore, like many other mixin-based languages [?, ?], McJava employs the structural subtyping approach.

The imported methods declared in the `requires` clause can be referred in the body of mixin. A mixin can overrides the imported methods and call the original method by using the `super` notation, e.g., `super.name()` called inside `Employee.name()`. In this case, we say that `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* [?, ?]. 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* [?], even though the template implemented by the mixin turns the method call structure upside down.

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 type 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 of `Id::Student::Employee`, so the latter is conceptually a subtype of the former. Therefore, the instance of latter can safely be type-casted to the former. This subtyping is a generalization of inheritance-based subtyping that adds much more flexibility to programs. This flexibility increases the chance of reusing programs that use composition types in their source code. If this subtyping is not allowed, an instance of `Id::Student::Employee` cannot be assigned type `Id::Employee`; therefore, some safe programs are rejected by the type checker.

In the above case, local variable `e` has type `Id::Employee`. The implementer of this method cannot predict what `e`'s *actual* type is; e.g. the implementer cannot predict `Id` will be composed with `Student`. If `Student.getID()` is called instead of `Employee.getID()` in the consequence of `super.getID()` in `ID`, the same problem of accidental overriding occurs. 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 s) {
  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, 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()`).

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()`.

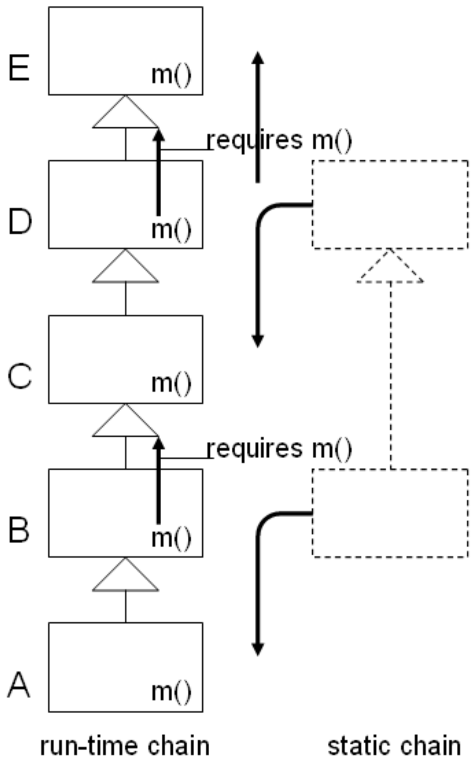

Figure 2: New method lookup in McJava

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()`].

Syntax:

$$
\begin{array}{rcl}
T & ::= & \bar{X} \mid \bar{X} :: \texttt{Object} \\
L_X & ::= & \texttt{mixin } X \texttt{ requires } I \\
 & & \quad \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\
L_I & ::= & \texttt{interface } I\ \{\ \bar{M_I};\ \} \\
K_X & ::= & X(\bar{T}\ \bar{f})\{\ \texttt{this.}\bar{f}\texttt{=}\bar{f}\texttt{;}\} \\
M & ::= & T\ m(\bar{T}\ \bar{x})\{\ \texttt{return } e;\} \\
M_I & ::= & T\ m(\bar{T}\ \bar{x}) \\
e & ::= & x \mid e.f \mid e.m\texttt{<}\bar{T}\texttt{>}(\bar{e}) \\
 & & \mid (\texttt{new } \bar{X} :: \texttt{Object}(\bar{e}), T) \\
 & & \mid \texttt{super[this]}.m\texttt{<}\bar{T}\texttt{>}(\bar{e}) \\
v & & (\texttt{new } \bar{X} :: \texttt{Object}(\bar{v}), T)
\end{array}
$$

Subtype relation:

$$T \ <:\ T \qquad \text{(S-REFL)}$$

$$\bar{X} :: X :: \bar{Y} \ <:\ \bar{X} :: \bar{Y} \qquad \text{(S-COMP)}$$

$$\frac{T \ <:\ S \qquad S \ <:\ U}{T \ <:\ U} \qquad \text{(S-TRANS)}$$

Figure 3: Syntax and subtype relation

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

4 FORMALIZATION

We have roughly sketched how the mechanism of selective method combination works. In this section, we propose a formal model of method combination of mixin-based subtyping to obtain concrete understanding of the proposed mechanism.

This formalization is based on Core McJava[?], a core language of McJava, that is an extension of FJ[?]. To focus on just a few key constructs of McJava type system, Core McJava provides tiny subset of McJava but captures key features of McJava such as mixin declarations, mixin composition, and mixin-based subtyping. The method combination mechanism introduced in this work depends on the `super` invocation and preservation of static type of receiver at run-time that are not included in Core McJava; therefore, this extension should include these features. We start with showing how the calculus extends the syntax of Core McJava.

Syntax.

The abstract syntax of the proposed calculus is given in Figure 3. In this paper, the metavariables d and e range over expressions; v and u range over values; K_X ranges over constructor declarations; m ranges over method names; M ranges over method declarations; X and Y range over mixin names; R, S, T, and U range over types; I ranges over interfaces; x and y range over variables; f and g range over field names.

As in Core McJava, we assume that the set of variables includes the special variable `this`, which is considered to be implicitly bound in every method declaration.

[illegible]

A method invocation expression is annotated with the static types $\bar{T}$ of m's arguments, written $e_0.m\texttt{<}\bar{T}\texttt{>}(\bar{e})$, which means this calculus provides method overloading. Method invocation on `super` is new; this feature is added to the calculus because it is crucial for the problem we are studying. This `super` invocation contains a variable `this` to hold the receiver of method invocation at run-time. Another change from Core McJava is, to preserve the static type of objects at run-time, the `new` expression is annotated with a static type T. This static type controls where the method lookup starts and may change while a reduction proceeds. We abbreviate (`new` $\bar{X}$ :: `Object`$(\bar{e}), \bar{X}$ :: `Object`) as `new` $\bar{X}$ :: `Object`$(\bar{e})$. In this calculus, class declarations are missing from the syntax, because in this paper we concentrate on understanding of the selective method combination mechanism in mixin-based subtyping.

We write $\bar{f}$ as a shorthand for a possibly empty sequence $f_1, \cdots, f_n$ and write $\bar{M}$ as a shorthand for $M_1 \cdots M_n$. The length of a sequence $\bar{x}$ is written as $\#(\bar{x})$. Empty sequences are denoted by $\cdot$. Similarly, we write "$\bar{T}\ \bar{f}$" as a shorthand for "$T_1\ f_1, \cdots, T_n\ f_n$"; "$\bar{T}\ \bar{f}$;" as a shorthand for "$T_1\ f_1; \cdots T_n\ f_n;$"; "`this`.$\bar{f} = \bar{f}$;" as a shorthand for "`this`.$f_1 = f_1; \cdots$`this`.$f_n = f_n;$"; $\bar{X}$ as a shorthand for $X_1 :: \cdots :: X_n$.

As mentioned above, there are no class declarations; therefore, we have only one class, `Object`, that is built-in the calculus. In this calculus, there are two kinds of types: $\bar{X}$ and $\bar{X}$ :: `Object`. The latter is a result of composing mixins (possibly empty sequence) and the class `Object`. Only the latter can be instantiated by using `new`.

We write $T\ <:\ U$ when T is a subtype of U. Subtype relations among compositions are defined in Figure 3, i.e., subtyping is a reflexive and transitive relation of the immediate subclass relation given by the composition subtyping rule.

Mixin table.

In this calculus, a program is a pair (XT, e) of a *mixin table* XT and an expression e. A mixin table is a map from mixin names to mixin declarations. The expression e may be considered as the `main` method of the "real" McJava program. The mixin table is assumed to satisfy the following conditions: (1) $XT(X) =$ `mixin` X ... for every $X \in dom(XT)$; (2)`Object` $\notin dom(XT)$; (3) $T \in dom(XT)$ for every mixin name appearing in the definition of any mixin in the table.

In the induction hypothesis, we abbreviate $XT(X) =$ `mixin` X ... as `mixin` X ....

Auxiliary functions.

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 4, 5 and 6.

$$\mathit{fields}(\texttt{Object}) = \cdot$$

$$\frac{\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\}}{\mathit{fields}(X) = \bar{T}\ \bar{f}}$$

$$\frac{\mathit{fields}(X) = \bar{T}\ \bar{f} \qquad \mathit{fields}(T) = \bar{S}\ \bar{g}}{\mathit{fields}(X :: T) = \bar{S}\ \bar{g},\ \bar{T}\ \bar{f}}$$

$$\frac{\mathit{fields}(T) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, T) = T_i}$$

Figure 4: Field lookup

$$\mathit{mtype}(m, \bar{T}, \texttt{Object}) = \texttt{nil}$$

$$\frac{\begin{array}{c}\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ S\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{\mathit{mtype}(m, \bar{S}, X) = S}$$

$$\frac{\begin{array}{c}\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ S\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \notin \bar{M}\end{array}}{\mathit{mtype}(m, \bar{S}, X) = \mathit{mtype}(m, \bar{S}, I)}$$

$$\frac{\texttt{interface}\ I\ \{\bar{M}_I;\} \qquad T\ m(\bar{T}\ \bar{x}) \in \bar{M}_I}{\mathit{mtype}(m, \bar{T}, I) = T}$$

$$\frac{\texttt{interface}\ I\ \{\bar{M}_I;\} \qquad T\ m(\bar{T}\ \bar{x}) \notin \bar{M}_I}{\mathit{mtype}(m, \bar{T}, I) = \texttt{nil}}$$

$$\frac{\mathit{mtype}(m, \bar{T}, X) = T}{\mathit{mtype}(m, \bar{T}, X :: T_0) = T}$$

$$\frac{\begin{array}{c}\mathit{mtype}(m, \bar{T}, X) = \texttt{nil} \\ \mathit{mtype}(m, \bar{T}, T_0) = T\end{array}}{\mathit{mtype}(m, \bar{T}, X :: T_0) = T}$$

Figure 5: Method type lookup

The fields of type T, given in Figure 4, written $\mathit{fields}(T)$, is a sequence $\bar{T}\ \bar{f}$ pairing the type of each field with its name. If T is a mixin, $\mathit{fields}(T)$ is a sequence for all the fields declared in that mixin. If T is a composition, $\mathit{fields}(T)$ is a sequence for all the fields declared in all of its constituent mixins. For the field lookup, we also have the definition of $\mathit{ftype}(f_i, T)$ that is a type of field f_i declared in T. Mixin composition rules described later will prevent field hiding.

The return type of method m declared in type T with argument types $\bar{T}$ is given by $\mathit{mtype}(m, \bar{T}, T)$. The function *mtype* is defined in Figure 5 by S that is a result type. If m with argument types $\bar{T}$ is not found in T, we define it `nil`. If T is a composition, the left operand of $::$ is searched first. The type of method m in interface I is also defined in the same way.

The body of method m declared in type T with argument types $\bar{T}$, given in Figure 6, written $\mathit{mbody}(m, \bar{T}, T, S)$, where m is a method name, $\bar{T}$ are argument types, T is the run-time type of receiver, S is the statically known type of receiver, is a triad, written $(\bar{x}, e, X)$, of a sequence of parameters $\bar{x}$, an expression e, and a mixin X where the resolved method is declared. The auxiliary function *mbody* formalizes the method lookup protocol roughly sketched in Figure 2. The first two rules assure that the leftmost m<$\bar{T}$> is selected when there are multiple candidates

$$\frac{mbody(m, \bar{T}, \bar{X} :: X, X) = (\bar{x}, e, Y)}{mbody(m, \bar{T}, \bar{X} :: X :: T, X[:: S]) = (\bar{x}, e, Y)}$$

$$\frac{\begin{array}{c} mbody(m, \bar{T}, \bar{X} :: X, X) = \texttt{nil} \\ mbody(m, \bar{T}, T, S) = (\bar{x}, e, Y) \end{array}}{mbody(m, \bar{T}, \bar{X} :: X :: T, X :: S) = (\bar{x}, e, Y)}$$

$$\frac{\downarrow mbody(m, \bar{T}, \bar{X}) = (\bar{x}, e, Y)}{mbody(m, \bar{T}, \bar{X} :: X, X) = (\bar{x}, e, Y)}$$

$$\frac{\begin{array}{c} \downarrow mbody(m, \bar{T}, \bar{X}) = \texttt{nil} \\ \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M} \end{array}}{mbody(m, \bar{T}, \bar{X} :: X[:: T], X) = (\bar{x}, e, X)}$$

$$\frac{\begin{array}{c} \downarrow mbody(m, \bar{T}, \bar{X}) = \texttt{nil} \\ \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \notin \bar{M} \end{array}}{mbody(m, \bar{T}, \bar{X} :: X, X) = \texttt{nil}}$$

$$\frac{\begin{array}{c} \downarrow mbody(m, \bar{T}, \bar{X}) = \texttt{nil} \\ \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \notin \bar{M} \\ \uparrow mbody(m, \bar{T}, T) = (\bar{x}, e, Y) \end{array}}{mbody(m, \bar{T}, \bar{X} :: X :: T, X) = (\bar{x}, e, Y)}$$

$$\frac{\begin{array}{c} \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M} \\ U\ m(\bar{T}\ \bar{x}) \in I \qquad \downarrow mbody(m, \bar{T}, \bar{X}) = \texttt{nil} \end{array}}{\downarrow mbody(m, \bar{T}, \bar{X} :: X) = (\bar{x}, e, X)}$$

$$\frac{\begin{array}{c} \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{y})\{\ \texttt{return}\ d;\ \} \in \bar{M} \\ U\ m(\bar{T}\ \bar{y}) \in I \\ \downarrow mbody(m, \bar{T}, \bar{X}) = (\bar{x}, e, Y) \end{array}}{\downarrow mbody(m, \bar{T}, \bar{X} :: X) = (\bar{x}, e, Y)}$$

$$\frac{\begin{array}{c} \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M} \\ U\ m(\bar{T}\ \bar{x}) \notin I \end{array}}{\downarrow mbody(m, \bar{T}, \bar{X} :: X) = \texttt{nil}}$$

$$\frac{\begin{array}{c} \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \notin \bar{M} \\ \downarrow mbody(m, \bar{T}, \bar{X}) = \bullet \\ \text{where } \bullet = (\bar{x}, e, Y) \text{ or } \texttt{nil} \end{array}}{\downarrow mbody(m, \bar{T}, \bar{X} :: X) = (\bar{x}, e, Y)}$$

$$\frac{\begin{array}{c} \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \notin \bar{M} \end{array}}{\uparrow mbody(m, \bar{T}, X) = \texttt{nil}}$$

$$\frac{\begin{array}{c} \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\ U\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M} \end{array}}{\uparrow mbody(m, \bar{T}, X) = (\bar{x}, e, X)}$$

$$\frac{\uparrow mbody(m, \bar{T}, X) = (\bar{x}, e, X)}{\uparrow mbody(m, \bar{T}, X :: T) = (\bar{x}, e, X)}$$

$$\frac{\begin{array}{c} \uparrow mbody(m, \bar{T}, X) = \texttt{nil} \\ \uparrow mbody(m, \bar{T}, T) = (\bar{x}, e, Y) \end{array}}{\uparrow mbody(m, \bar{T}, X :: T) = (\bar{x}, e, Y)}$$

Figure 6: Method body lookup

(We assume that mixin X is not used in the composition $\bar{X}$. The notation $X[:: T]$ represents that the occurrence of composition is optional). The following three rules trigger the subfunction $\downarrow$ *mbody* that formalizes the *down search* of the run-time types. If $\downarrow$ *mbody* defines `nil`, which means that the down search fails, and if method lookup on current mixin X also fails, the *mbody* function defines it `nil`. The last rule of *mbody* is used for super calls. The first two rules of $\downarrow$ *mbody* assures that intentional overriding is allowed; the third rule forbids accidental overriding. Another subfunction $\uparrow$ *mbody* searches *up* the run-time types, as the pure Java's method lookup protocol does.

Typing.

The typing rules for compositions and expressions are given in Figure 7. A composition is well-formed if (1) there are no fields declared with the same name both in the left component and the right component of the composition, (2) there is no method collision, that is, if some methods are declared with the same name and with the same argument types in the left and the right, the return type of both methods must be the same (that is expressed by the VALIDOVERRIDE predicate in Figure 7 corresponding to the Java rule on *intentional* overriding), and (3) for all the methods declared in the interface that is required by the left mixin, the right operand of the composition declares the methods named and typed as the same as the interface.

An environment Γ is a finite mapping from variables to types, written $\bar{x} : \bar{T}$. The typing judgment for expressions has the form $\Gamma \vdash e : T$, read "in the environment Γ, expression e has type T". The typing rules for constructor and method invocations check that the type of each argument is a subtype of the corresponding formal parameter. T-SUPER assures that a `super` invocation can be typed only when the type of e, a receiver of the `super` invocation, is a subtype of enclosing mixin (as explained above, the syntax assumes that e is `this`, but `this` is substituted with a value while reductions proceed).

Figure 7 also shows the rules for well-formed definitions of methods and mixins. The type of the body of a method declaration is a subtype of the declared type. VALIDSUPER forbids `super` calls on objects other than the caller. A mixin is well-formed if all the methods declared in that mixin are well-formed.

Dynamic semantics.

There are not many drastic changes in typing rules from Core McJava. We have to change, however, the dynamic semantics in a significant way.

The reduction relation is of the form $e \longrightarrow e'$, read "expression e reduces to expression e' in one step". We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

Auxiliary definitions:

$$\frac{\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ \ldots\ \}}{\mathit{interface}(X) = I}$$

$$\frac{\begin{array}{l}\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ \ldots\ \ \bar{M}\ \}\\ \forall(S\ m(\bar{T}\ \bar{x})\{\ldots\}) \in \bar{M}\\ \quad \mathit{mtype}(m, \bar{T}, X) = \mathit{mtype}(m, \bar{T}, T)\ \text{or}\\ \quad \mathit{mtype}(m, \bar{T}, T) = \texttt{nil}\end{array}}{\text{VALIDOVERRIDE}(X, T)}$$

$$\frac{\begin{array}{c}\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ \ldots\ \}\\ \texttt{interface}\ I\ \{\bar{M}_I\}\\ \text{If } T \text{ is a composition } \bar{X} :: \texttt{Object}, \text{ then}\\ \forall(U\ n(\bar{S}\ \bar{x})) \in \bar{M}_I\\ \mathit{mtype}(n, \bar{S}, I) = \mathit{mtype}(n, \bar{S}, T)\end{array}}{\text{COMPOSITIONCOMPLETE}(X, T)}$$

$$\frac{\begin{array}{l}\text{If } e \text{ contains } \texttt{super}[d].m\texttt{<}\bar{T}\texttt{>}(\bar{e}), \text{ then}\\ \quad d = \texttt{this}\end{array}}{\text{VALIDSUPER}(e)}$$

Well-formed composition:

$$\frac{\begin{array}{c}\mathit{fields}(X) \cap \mathit{fields}(T) = \emptyset\\ \text{VALIDOVERRIDE}(X, T)\\ \text{COMPOSITIONCOMPLETE}(X, T)\end{array}}{X :: T\ \text{ok}} \quad \text{(T-COMP)}$$

Expression typing:

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad \mathit{ftype}(f, S) = T}{\Gamma \vdash e_0.f : T} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : S \qquad \mathit{mtype}(m, \bar{S}, S) = T\\ \Gamma \vdash \bar{e} : \bar{T} \qquad \bar{T}\ <:\ \bar{S}\end{array}}{\Gamma \vdash e_0.m\texttt{<}\bar{S}\texttt{>}(\bar{e}) : T} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\bar{X} :: \texttt{Object}) = \bar{S}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{T}\\ \bar{T}\ <:\ \bar{S} \qquad \bar{X} :: \texttt{Object}\ \text{ok}\\ \bar{X} :: \texttt{Object}\ <:\ T\end{array}}{\Gamma \vdash (\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{e}), T) : T} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\mathit{mtype}(m, \bar{S}, \mathit{interface}(X)) = T \qquad \Gamma \vdash e : T_0\\ \Gamma \vdash \bar{e} : \bar{T} \qquad \bar{T}\ <:\ \bar{S} \qquad T_0\ <:\ X\end{array}}{\Gamma \vdash \texttt{super}[e].m\texttt{<}\bar{S}\texttt{>}(\bar{e}) : T} \quad \text{(T-SUPER)}$$

Well-formed definitions:

$$\frac{\begin{array}{l}\bar{x} : \bar{T}, \texttt{this} : X \vdash e_0 : S_0 \qquad S_0\ <:\ T_0\\ T_0\ \text{ok} \qquad \bar{T}\ \mathit{ok} \qquad \text{VALIDSUPER}(e_0)\\ \qquad \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ldots\}\end{array}}{T_0\ \texttt{m}(\bar{T}\ \bar{x})\{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ X} \quad \text{(T-XMETHOD)}$$

$$\frac{\begin{array}{c}K_X = X(\bar{T}\ \bar{f})\{\ \texttt{this}.\bar{f}\texttt{=}\bar{f};\}\\ \bar{M}\ \texttt{OK IN}\ X \qquad \bar{T}\ \text{ok}\end{array}}{\texttt{mixin}\ X\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\}\ \texttt{OK}} \quad \text{(T-MIXIN)}$$

Figure 7: Typing rules

The reduction rules are given in Figure 8. We write $[((\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{e}), S)/x)_T]e_0$ for the result of substituting x by $(\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{e}), T)$ in e_0. T is the type of variable x. Note that after this substitution, the static type associated with the value, which is used for method body lookup, is changed from S to T[1]. We write "$(\bar{v}/\bar{x})_{\bar{T}}$" as a shorthand for "$(v_1/x_1)_{T_1}, \cdots, (v_n/x_n)_{T_n}$."

There are three reduction rules, one for field access, one for method invocation, and one for method invocation on `super`. The field access reduces to the correspond-

[1]The dynamic type is unchanged to handle intentional overriding.

$$\frac{\mathit{fields}(\bar{X} :: \texttt{Object}) = \bar{T}\ \bar{f}}{(\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{v}), S).f_i \longrightarrow v_i} \qquad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(m, \bar{T}, \bar{X} :: \texttt{Object}, S) = (\bar{x}, e_0, X)}{\begin{array}{c}(\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{v}), S).m\texttt{<}\bar{T}\texttt{>}(\bar{u}) \\ \longrightarrow [(\bar{u}/\bar{x})_{\bar{T}}, ((\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{v}), S)/\texttt{this})_X]e_0\end{array}} \qquad \text{(R-INVK)}$$

$$\frac{\begin{array}{c}\mathit{mbody}(m, \bar{T}, S, S') = (\bar{x}, e_0, X) \\ \text{where } S' = \mathit{super}(\bar{X} :: \texttt{Object}, S)\end{array}}{\begin{array}{c}\texttt{super}[(\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{v}), S)].m\texttt{<}\bar{T}\texttt{>}(\bar{u}) \\ \longrightarrow [(\bar{u}/\bar{x})_{\bar{T}}, ((\texttt{new}\ \bar{X} :: \texttt{Object}(\bar{v}), S)/\texttt{this})_X]e_0\end{array}}$$
(R-INVK-SUPER)

where

$$\mathit{super}(\bar{X} :: X :: \bar{T}, X) = \bar{T}$$

$$\frac{e_0 \longrightarrow e_0^{'}}{e_0.f \longrightarrow e_0^{'}.f} \qquad \text{(RC-FIELD)}$$

$$\frac{e_0 \longrightarrow e_0^{'}}{e_0.m\texttt{<}\bar{T}\texttt{>}(\bar{e}) \longrightarrow e_0^{'}.m\texttt{<}\bar{T}\texttt{>}(\bar{e})} \qquad \text{(RC-INVK-RECV)}$$

$$\frac{e_i \longrightarrow e_i^{'}}{e_0.m\texttt{<}\bar{T}\texttt{>}(\bar{v}, e_i, \bar{e}) \longrightarrow e_0.m\texttt{<}\bar{T}\texttt{>}(\bar{v}, e_i^{'}, \bar{e})} \qquad \text{(RC-INVK-ARG)}$$

$$\frac{e_i \longrightarrow e_i^{'}}{(\texttt{new}\ \bar{X} :: C(\bar{v}, e_i, \bar{e}), T) \longrightarrow (\texttt{new}\ \bar{X} :: C(\bar{v}, e_i^{'}, \bar{e}), T)} \qquad \text{(RC-NEW)}$$

$$\frac{e_i \longrightarrow e_i^{'}}{\texttt{super}[e].m\texttt{<}\bar{T}\texttt{>}(\bar{v}, e_i, \bar{e}) \longrightarrow \texttt{super}[e].m\texttt{<}\bar{T}\texttt{>}(\bar{v}, e_i^{'}, \bar{e})} \qquad \text{(RC-SUPER)}$$

Figure 8: Operational semantics

ing argument for the constructor. Due to the stylized form of object constructors, the constructor has one parameter for each field, in the same order as the fields are declared. The method invocation reduces to the expression of the method body, substituting all the parameter $\bar{x}$ with the argument expressions $\bar{d}$ and the special variable `this` with the receiver. While the substitution, the static type of argument is replaced with the corresponding parameter type. The method invocation on `super` also reduces to the method body.

Properties.

The proof of type soundness is almost the same as that of Core McJava [?], except that we should also consider the induction cases for T-SUPER and R-INVK-SUPER.

Lemma 4.1 *If ftype*$(f, U) = T$*, then ftype*$(f, S) = T$ *for all* $S\ <:\ U$.

Proof. Straightforward induction on the derivation of subtype relation <: and *ftype*. □

Lemma 4.2 *If mtype*$(m, \bar{T}, U) = T_0$*, then mtype*$(m, \bar{T}, T) = T_0$ *for all* $T\ <:\ U$.

Proof. Straightforward induction on the derivation of subtype relation <:, *mtype* and T-COMP. □

Lemma 4.3 *If* $\Gamma, \bar{x} : \bar{S} \vdash e : U$, $\Gamma \vdash \bar{d} : \bar{R}$ *where* $\bar{R}\ <:\ \bar{S}$*, then* $\Gamma \vdash [(\bar{d}/\bar{x})_{\bar{T}}]e : T$ *for some* $T\ <:\ U$ *and* $\bar{R}\ <:\ \bar{T}\ <:\ \bar{S}$.

Proof. By induction on the derivation of $\Gamma, \bar{x} : \bar{S} \vdash e : U$, Lemma 4.1 and 4.2. □

Lemma 4.4 *If* $\Gamma \vdash e : T$ *where* Γ *does not include* x*, then* $\Gamma, x : U \vdash e : T$.

Proof. Straightforward induction. □

Lemma 4.5 *If mtype*$(m, \bar{U}, \bar{X}) = U$ *and mbody*$(m, \bar{U}, R, \bar{X}) = (\bar{x}, e, X)$ *where* $R\ <:\ \bar{X}$*, then, for some* U_0 *with* $\bar{X}\ <:\ U_0$*, there exists* $T\ <:\ U$ *such that* $\bar{x} : \bar{U}, \texttt{this} : U_0 \vdash e : T$.

Proof. By induction on the derivation of *mbody*. □

Theorem 4.1 (Subject Reduction) *If* $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$*, then* $\Gamma \vdash e' : T'$ *for some* $T'\ <:\ T$.

Proof. By induction on a derivation of $e \longrightarrow e'$, Lemma 4.3, 4.4 and 4.5. □

Theorem 4.2 (Progress) *Suppose e is a well-typed expression.*

1. *If* e *includes* $(\texttt{new}\ \bar{X} :: \texttt{Object}, S)(\bar{e}).f$ *as a subexpression, then fields*$(\bar{X} :: \texttt{Object}) = \bar{T}\ \bar{f}$ *and* $f \in \bar{f}$ *for some* $\bar{T}$ *and* $\bar{f}$.
2. *If* e *includes* $(\texttt{new}\ \bar{X} :: \texttt{Object}, S)(\bar{e}).m\texttt{<}\bar{T}\texttt{>}(\bar{d})$ *as a subexpression, then mbody*$(m, \bar{T}, \bar{X} :: \texttt{Object}, S) = (\bar{x}, e_0, X)$, $\emptyset \vdash \bar{d} : \bar{S}$ *where* $\bar{S}\ <:\ \bar{T}$*, and* $\#(\bar{x}) = \#(\bar{d})$ *for some* $\bar{x}$ *and* e_0.

3. *If e includes* `super`$[($`new` $\bar{X}$ :: `Object`$, S)].m$<$\bar{T}$>$(\bar{d})$ *as a subexpression, then* $mbody(m, \bar{T}, S, S') = (\bar{x}, e_0, X)$*,* $\emptyset \vdash \bar{d} : \bar{S}$*,* $S' = super(\bar{X}$ :: `Object`$, S)$ *where* $\bar{S} <: \bar{T}$*, and* $\#(\bar{x}) = \#(\bar{d})$ *for some* $\bar{x}$ *and* e_0*.*

Proof. Immediate from well-typedness of the subexpression. □

Theorem 4.3 (Type Soundness) *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow^* e'$ *with* e' *a normal form, then* e' *is a value* v *of* e *with* $\emptyset \vdash v : U$ *and* $U <: T$*.*

Proof. Immediate from Theorem 4.1 and 4.2. □

5 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.

Figure 9 shows the translated Java code from the definitions in Figure 1 and `Id` in section 2. In the following subsections, we show the compilation strategy in detail.

Inheritance chain translation

McJava compilation strategy is explained by Kamina and Tamai in [?, ?]. At first, McJava mixin composition is translated into Java's class hierarchy:

- A composition $\mathtt{X}_1$`::`$\cdots$`::`$\mathtt{X}_n$`::C`, where each $\mathtt{X}_i$ $(i \in 1 \cdots n)$ is a mixin and `C` is a class, is translated into a class $\mathtt{X}_1\_\cdots\_\mathtt{X}_n\_\mathtt{C}$ that implements the interface $\_\mathtt{X}_1\_\cdots\_\mathtt{X}_n\_\mathtt{C}$ and extends the class $\mathtt{X}_2\_\cdots\_\mathtt{X}_n\_\mathtt{C}$. The body of the class $\mathtt{X}_1\_\cdots\_\mathtt{X}_n\_\mathtt{C}$ is copied from $\mathtt{X}_1$. The interface $\_\mathtt{X}_1\_\cdots\_\mathtt{X}_n\_\mathtt{C}$ extends all the interfaces that correspond to each of $\mathtt{X}_1$`::`$\cdots$`::`$\mathtt{X}_n$`::C`'s immediate super types.

  For example, `Id::Student::Employee::Person` is translated into a class `Id_Student_Employee_Person` that extends a class `Student_Employee_Person` and implements an interface `_Student_Employee_Person`. The class `Student_Employee_Person` extends a class `Employee_Person` and implements an interface `_Employee_Person`. The interface `_Student_Employee_Person` extends `_Student_Employee`, `_Student_Person`, and `_Employee_Person`.

- All the composition types that appear in class definitions and interface definitions are replaced with corresponding interface names. Similarly, all the

```
class Person {
  String _name;
  String name() { return _name; }
}
interface _Employee {
  String name();
  String Employee$getID();
}
interface _Employee_Person extends _Employee, _Person {}
class Employee_Person extends Person
      implements _Employee_Person {
  String id, title;
  String name() { return title+super.name(); }
  String Employee$getID() { return id; }
}
interface _Student { String Student$getID(); }
interface _Student_Employee_Person
    extends _Student_Employee,_Student_Person,_Employee_Person
  { }
class Student_Employee_Person extends Employee_Person
      implements _Student_Employee_Person {
  String id;
  String Student$getID() { return id; }
}
interface Id { String getID(); ...; }
interface _Id_Student_Employee_Person
    extends _Id_Student_Employee, _Id_Student_Person,
            _Id_Employee_Person, _Student_Employee_Person
  { }
class Id_Student_Employee_Person
      extends Student_Employee_Person
      implements _Id_Student_Employee_Person {
  String Student$getID() {
    return super.Student$getID(); }
  String Employee$getID() {
    return super.Employee$getID(); }
  String getID() {
    return super.Student$getID(); }
  ...
}
```

Figure 9: Compiled code from Figure 1 and `Id` (Some details are omitted for the reason of limited space)

composition constructor invocations that appear in class definitions are replaced with corresponding class names.

For example, the statement

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

is translated into the following statement:

```
_Id_Student_Employee p =
  new Id_Student_Employee_Person()
```

Renaming strategy

To preserve the behavior of selective method combination in translated Java programs, methods are renamed while the compilation.

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$getID()` in the translated class (`Id_Employee_Student_Person`). If there exist multiple method combinations, the compiler also inserts new methods whose names are the same as those of overridden methods, copying body of the overriding method. For example, the method declaration `getID()` in `Id` is also copied into the method declaration of `Employee$getID()` in the translated class. Note that the name of the method in method invocation on `super` is also changed appropriately.

The method name invoked externally 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();
  ...
}
```

6 RELATED WORK

As mentioned earlier, our approach is an extension of hygienic mixins [?, ?]. The implementation of hygienic mixins is based on MixedJava, formalized by Flatt et al. [?]. 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 [?, ?, ?]. Note that MixedJava employs nominal subtyping for composition checking.

Ernst proposed the *propagation* mechanism of method combination in the statically typed language `gbeta` [?], a generalization of the language BETA [?]. `gbeta` also provides similar mechanism with our approach that allows two methods with the same signature to coexist 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 [?]. We also note that recently Goldberg et al. propose a language that integrates `super` and `INNER` [?].

Traits [?] 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 [?, ?, ?] 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.

7 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. The formalization promotes understanding of the proposed system. 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.

Acknowledgments: 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.

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] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[9] Tetsuo Kamina. *A Design and Implememtation of Mixin-Based Composition in Strongly Typed Object-Oriented Languages.* PhD thesis, The University of Tokyo, 2005.

[10] Tetsuo Kamina and Tetsuo Tamai. A core calculus for mixin-types. In *Foundations on Object Oriented Languages (FOOL11)*, 2004. Revised version is available at `http://www.graco.c.u-tokyo.ac.jp/~kamina/papers/fool/kamina.pdf`.

[11] 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.

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

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

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

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

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

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

[18] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *ICSE 2005*, pages 166–175, 2005.

[19] 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.

Tetsuo Kamina received his B.A. in International Christian University, M.A. and Ph.D. in the University of Tokyo. He became a JST CREST researcher of Center for Tsukuba Advanced Research Alliance (TARA), University of Tsukuba in 2005 and has been in that position. His current research includes object-oriented language design and implementation, software engineering, and signal processing especially for still image processing. He can be reached at kamina@acm.org.

Tetsuo Tamai received the B.S., M.S. and Dr.S. degrees in mathematical engineering from the University of Tokyo.
He joined Mitsubishi Research Institute, Inc. in April 1972 and had been the manager of Artificial Intelligence Technologies Section from October 1985 to March 1989. He became an Associate Professor of Graduate School of Systems Management, the University of Tsukuba in 1989. He then became a Professor of Graduate School of Arts and Sciences, the University of Tokyo in 1994 and has been in that position ever since. His current research includes high reliability component-based software engineering, collaboration and role modeling, formal analysis of software architectures and software evolution process.
He has been contributing to the activities of Japan Society for Software Science and Technology for a long time as a board member and as the Editor-in-Chief of its journal "Computer Software." He served as the Program Chair of JSSST 20th anniversary conference in September 2003. He is also on the editorial board of "Information and Software Technology", published by Elsevier Science. He is currently a member of the executive committee of ACM SIGSOFT as an International Liaison. He was also a past chair of Special Interest Group on Software Engineering, Information Processing Society of Japan and a past chair of the Software Engineers Association, Japan.
He has been sharing responsibilities of a number of international academic conferences, including PC of ICSE's, RE's, ESEC/FSE's, ICSM's and many others and Steering Committee of APSEC and IWPSE.