McJava – A Design and Implementation of Java with Mixin-Types

Tetsuo Kamina and Tetsuo Tamai

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

Abstract. A programming construct *mixin* was invented to implement uniform extensions and modifications to classes. Although mixin-based programming has been extensively studied both on the methodological and theoretical point of views, relatively few attempts have been made on designing *real* programming languages that support mixins. In this paper, we address the issue of how to introduce a feature of declaring a mixin that may also be used as a type to nominally typed object-oriented languages like Java. We propose a programming language McJava, an extension of Java with mixin-types. To study type-soundness of McJava, we have formulated the core of McJava with typing and reduction rules, and proved its type-soundness. We also describe a compilation strategy of McJava that translates McJava programs to Java programs thus eventually making it runnable on standard Java virtual machines.

1 INTRODUCTION

Object-oriented programming languages like Java and C# offer class systems that provide a simple and flexible mechanism for reusing collections of program pieces. Using inheritance and overriding, programmers may derive a new class by specifying only the elements that are extended and modified from the original class. However, a pure class-based approach lacks a mechanism of abstracting uniform extensions and modifications to multiple classes.

A programming construct *mixin* (also known as *abstract subclass*) was invented to implement modules that provide such uniform extensions and modifications [20]. This construct provides much reusability because a mixin makes it possible to add common features (that will be duplicated in a single inheritance hierarchy) to a variety of classes. Mixin-based programming, popularized by CLOS [17], has been studied both on the methodological and theoretical point of views [8, 9, 4, 7, 12]. Small core languages that support mixins or *mixin modules* are also proposed [13, 11, 16]. Despite the existence of these extensive studies, relatively few attempts are made on designing *real* programming languages that support mixins with notable exception of the language Jam [3] that integrates mixins with Java.[1]

[1] We will note differences between Jam and our approach in section 5.

In many cases, a mixin is considered as a means for providing uniform extension to classes; however, some mixin-based systems also allow a mixin to be composed with other mixins (e.g. [13]). Composition of two mixins produces another mixin that has both features of its constituents. It can be regarded as a kind of inheritance in the form of mixin composition, which enhances the reusability of mixins still further.

In this paper, we address how to add *mixin-types*, a mechanism of declaring a mixin that may also be used as a type, to nominally typed, mainstream object-oriented languages like Java. We present a programming language McJava.[2] McJava has the following features; (1) mixins are explicitly supported as a language feature, and mixin names are used as types; (2) *higher order mixins* (mixins composed with other mixins) are supported, and flexible subtyping rules among subsequences of composition are provided, that promotes much flexible code reuse; (3) mixin composition is a subject to type-checking.

To study type-soundness of McJava, we have developed Core McJava, a small calculus for McJava that is based on FJ [15], a tiny subset of Java. Because Core McJava is very small, it is suitable for focusing on type-checking issues. We have proved the type-soundness theorem of Core McJava, that provides an assurance that McJava type system is sound.

Because McJava is designed as an extension of Java, it is desirable that McJava programs may run on the standard JVM; however, owing to its flexibility of subtyping, how to compile McJava programs to JVM is not so straightforward. Because Java does not allow a class to inherit from multiple classes, McJava subtyping must be *linearized* in the compilation. This linearization imposes unnecessarily deep inheritance chains to the compiled program. In this paper, we also present a compilation strategy of McJava, and show an optimization algorithm by eliminating unused types from the inheritance chains. Based on this strategy, we implemented a prototype version of McJava compiler that type-checks McJava programs and translates them into Java programs thus making it runnable on the standard JVM.

We summarize the contributions of this paper:

- Introducing mixins into a mainstream, statically-typed language.
- Including higher order mixins and mixin-based subtyping.
- Establishing the soundness of the type system.
- Devising a compilation strategy and optimization.

2 AN OVERVIEW OF MCJAVA

Mixin declarations and mixin-types. To demonstrate how a mixin is declared in McJava, we start with a very simple example. Figure 1 shows a declaration of mixin `Color`. This mixin provides "color" feature that is intended to be composed with widget classes.

A statement beginning with a keyword `mixin` is a *mixin declaration.* A mixin declaration has the following form:

[2] **M**ixin-based **C**ompositions for **Java**.

```
interface WidgetI { void paint(Graphics g); }
mixin Color requires WidgetI {
  int color;
  void paint(Graphics g) {
    g.setColor(this);
    super.paint(g);
    ... }
  void setColorValue(int color) { this.color=color; }
  int getColorValue() { return this.color; }
}
```

Fig. 1. A color mixin

`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, both class `Label` and class `TextField`, declared as

```
class Label implements WidgetI { void paint(Graphics g) { .. }}
class TextField { void paint(Graphics g) { ... } }
```

can be composed with mixin `Color`, as they implement interface `WidgetI`. Note that, it is not necessary for these classes to explicitly declare that they implement interface `WidgetI`, as shown by class `TextField`. A class that implicitly implements a `paint` method (i.e. a class that has a `void paint()` method without declaring `implements WidgetI`) may also be composed with mixin `Color`.[3]

When a required interface is declared in a mixin, methods are to be imported to the mixin from a class to be composed. For example, the `paint` method in mixin `Color` invokes `super.paint(g)` that results in invocation of `paint` declared in `Color`'s "superclass". McJava also allows an anonymous interface to appear in `requires` clause for more handy syntax:

```
mixin Color requires { void paint(graphics g); } { ... }
```

If a mixin requires *no* interfaces (i.e. a mixin that imports no methods), we may omit the `requires` clause.

A composition of mixin `Color` and class `Label` is written as `Color::Label`. This composition is regarded as a subclass that is derived from the parent `Label` class, with subclass body declarations being the same as the body of mixin `Color`. Similarly, a composition `Color::TextField` is regarded as a subclass of `TextField`. In this sense, a mixin is a uniform extension of classes that may be applied to many different parent classes.

[3] Note that the `requires` clause of a mixin declaration is quite different from `implements` clause of ordinary class declarations in that a required interface in mixin declaration is not used as a type but used as a *constraint*. In fact, there is no subtype relation between mixin `Color` and interface `WidgetI`.

```
interface WidgetI { void paint(Graphics g); }
mixin Font requires WidgetI {
  String font;
  void paint(Graphics g) {
    g.setFont(this);
    super.paint(g);
    ... }
  void setFontName(String font) { this.font=font; }
  String getFontName() { return this.font; }
}
```

Fig. 2. A Font mixin

Besides this modularity, McJava also provides the useful feature of mixin-types, which means a declared mixin is also used as a type. It is to be noted that using a mixin as a type is often useful to abstract all the results of composing the mixin with other classes and mixins. We may write the name `Color`, for example, in a formal parameter of a method declaration that results in a method that takes an instance of all the results of composing mixin `Color` with composable classes as an argument.

As an abstract class in Java cannot be instantiated, it is forbidden to create an instance of an abstract subclass (i.e. a mixin) in McJava.

Higher order mixins and subtyping. In McJava a mixin may also be composed with a mixin. For example, the previous mixin `Color` may be composed with mixin `Font` declared in Figure 2. This composition, written as `Color::Font`, is regarded as a mixin that has both features of `Color` and `Font`.

A mixin `Color` may also be composed with a composition `Font::Label` resulting in a new composition `Color::Font::Label`. The composition operator `::` is associative, that is a result of composing a mixin `Color` with a composition `Font::Label`, written `Color::(Font::Label)`, is the same as `(Color::Font)::Label`, a result of composing `Color::Font` with `Label` (recall that a composition of a mixin and another mixin is also regarded as a mixin).

A composition `Color::Font::Label` provides all the methods declared in `Color`, `Font`, and `Label`. In McJava, the order of method lookup for compositions is well-defined. If a method `paint` is searched on `Color::Font::Label`, for instance, `Color` is searched first, then `Font`, followed by `Label`. Because the order of method lookup controls the *behavior* [18] of mixin compositions, the composition operator `::` is not commutative. For instance, `Color::Font` is not the same type as `Font::Color`, because the behavior of each composition may be different.

One of the novel features of McJava is the flexibility of its subtype relation over compositions. In McJava, a composition is a subtype of all its constituent. For example, `Color::Font::Label` is a subtype of `Label`, `Font`, and `Color`. It is also a subtype of its subsequences, `Font::Label`, `Color::Font` and

(maybe somewhat surprisingly) `Color::Label`. Because the operator `::` is not commutative, the order of composition is significant (i.e. `Color::Font` is not a subtype of `Font::Color`). The further reason of this restriction is, if we do not require respecting order in subtyping between sequences, `Color::Font` is a subtype of `Font::Color` that is a subtype of `Color::Font`. This means subtype relation is no longer partial order because, as mentioned earlier, `Color::Font` $\neq$ `Font::Color`, which will confuse many Java users. However, it is interesting to investigate whether the type system remains sound with this more flexible definition of composition subtyping. This issue remains as one of our future work.

The subtyping system proposed here enhances much reusability of code. Consider the situation where we use normal Java to extend a class `Label` to `FontLabel`, then further to `ColorFontLabel`. Suppose, we also extend `Label` to `ColorLabel` independently. In Java, however, `ColorFontLabel` is not a subtype of `ColorLabel`.

Mixin composability. Adding mixin-types to Java type system requires the type-checker to perform more sophisticated type-checking. We briefly summarize here what McJava type-checker does to check the well-typedness of mixin compositions. To ensure that compiled McJava programs run safely, the type-checker must check whether the following requirements are met:

- For all the compositions $\mathtt{X}_1\mathtt{::}\cdots\mathtt{::X}_n\mathtt{::C}$, where $\mathtt{X}_1, \cdots, \mathtt{X}_n$ are mixins and `C` is a class, the composition $\mathtt{X}_2\mathtt{::}\cdots\mathtt{::C}$ must implement all the interfaces that the mixin $\mathtt{X}_1$ requires.
- For all the compositions `X::T`, where `X` is a mixin and `T` is a mixin, a class, or a composition, if `X` declares a method *m* and a method *m'* with the same name *m* and the same signature is also declared in `T`, then the return type of *m* must be the same as the type of *m'*.

The first rule ensure that no "method not understood" error occurs at run-time. The second rule corresponds to the Java rule on overriding. In other words, if the mixin `X` accidentally "overrides" a method declared in `T` with the different return type, the compiler reports an error.

3 CORE CALCULUS OF MCJAVA

To provide an assurance that McJava type system is sound, we have developed Core McJava, a small calculus of McJava that is suitable for proving the type soundness theorem.

The design of Core McJava is based on FJ [15], a minimum core language of Java. FJ is a very small subset of Java, focusing on just a few key constructs. For example, FJ constructors always take the same stylized form: there is one parameter for each field, with the same name as the field. FJ provides no side-effective operations, that means a method body always consists of `return` statement followed by an expression. Because FJ provides no side-effects, the only place where

$$
\begin{aligned}
T &::= \bar{X} :: C \mid \bar{X} \\
L_C &::= \texttt{class}\ C\ \texttt{extends}\ \bar{X} :: C \\
&\qquad \{\bar{T}\ \bar{f};\ K_C\ \bar{M}\} \\
L_X &::= \texttt{mixin}\ X\ \texttt{requires}\ I \\
&\qquad \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\} \\
L_I &::= \texttt{interface}\ I\ \{\ \bar{M}_I;\ \} \\
K_C &::= C(\bar{S}\ \bar{g},\ \bar{T}\ \bar{f}) \\
&\qquad \{\texttt{super}(\bar{g});\ \texttt{this}.\bar{f}\texttt{=}\bar{f};\} \\
K_X &::= X(\bar{T}\ \bar{f})\{\ \texttt{this}.\bar{f}\texttt{=}\bar{f};\} \\
M &::= T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\} \\
M_I &::= T\ m(\bar{T}\ \bar{x}) \\
e &::= x \mid e.f \mid e.m\texttt{<}\bar{T}\texttt{>}(\bar{e}) \\
&\quad \mid \texttt{new}\ \bar{X} :: C(\bar{e})
\end{aligned}
$$

Fig. 3. Core McJava syntax

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

$$
\begin{aligned}
T_1 :: \cdots :: T_n &<:\ T_2 :: T_3 :: \cdots :: T_n \\
&<:\ T_1 :: T_3 :: \cdots :: T_n \\
&\cdots \\
&<:\ T_1 :: T_2 :: \cdots :: T_{n-1}
\end{aligned}
\qquad \text{(S-COMP)}
$$

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

$$\frac{\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\ldots\}}{C\ <:\ \bar{X} :: D} \qquad \text{(S-CLASS)}$$

Fig. 4. Subtype relation

assignment operations may appear is within a constructor declaration. In FJ, all the fields are initialized at the object instantiation time. Once initialized, an FJ object never changes its state. FJ does not support modifiers of members and constructors, that means all the members and constructors of classes are public. Interfaces are also not supported by FJ.

Core McJava shares the same features of FJ explained above. In the following subsections, we present the syntax and operational semantics of Core McJava and its type soundness theorem.

Syntax. The abstract syntax of Core McJava is given in Figure 3. In this paper, the metavariables d and e range over expressions; K_C and K_X range over constructor declarations; m ranges over method names; M ranges over method declarations; C and D range over class names; X and Y range over mixin names; R, S, T, U and V range over type names; I ranges over interface names; x ranges over variables; f and g range over field names. As in FJ, we assume that the set of variables includes the special variable `this`, which is considered to be implicitly bound in every method declaration. Unlike full McJava, and as in FJ, Core McJava does not allow classes to implement interfaces; however, Core McJava provides interfaces that are used only in the `requires` clause. This is a primary feature of McJava that cannot be excluded from the core calculus.

In Core McJava, a method invocation expression $e_0.m(\bar{e})$ is annotated with the static types $\bar{T}$ of m's arguments, written $e_0.m\texttt{<}\bar{T}\texttt{>}(\bar{e})$. This annotation is necessary because, unlike FJ, Core McJava actually provides method overloading. To capture the McJava's feature of overloaded method resolution, determining which overloaded method to invoked at compile time, a method invocation expression necessarily retains the static types of its arguments. We include this

feature in Core McJava, because it is crucial for the problem we are studying.[4] Because of this condition, Core McJava is not a subset of McJava whereas FJ is a subset of Java; instead, we view Core McJava as an intermediate language to which the user's programs in McJava are translated. This translation is straightforward.

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 "$T_1\ f_1; \cdots T_n\ f_n;$"; "$\texttt{this}.\bar{f} = \bar{f};$" as a shorthand for "$\texttt{this}.f_1 = f_1; \cdots \texttt{this}.f_n = f_n;$"; $\bar{X}$ as a shorthand for $X_1 :: \cdots :: X_n$.

As in Figure 3, there are two kinds of types: $\bar{X}$ and $\bar{X} :: C$. The former denotes a *mixin-mixin composition* that is generated by composing mixin names, while the latter denotes *mixin-class composition* that is a result of composing mixin names (possibly empty sequence) and a class name. The former is a mixin that cannot be instantiated, while the latter is a concrete class that can be instantiated.

We write $T \ <: \ U$ when T is a subtype of U. Subtype relations between classes, mixins, and compositions are defined in Figure 4, i.e., subtyping is a reflexive and transitive relation of the immediate subclass relation given by the `extends` clauses in class declarations and mixin compositions.

Class table. A Core McJava program is a pair of (CT, e) of a *class table* CT and an expression e. A class table is a map from class names and mixin names to class declarations and mixin declarations. The expression e may be considered as the `main` method of the "real" McJava program. The class table is assumed to satisfy the following conditions: (1) $CT(C) = \texttt{class}\ C\ \ldots$ for every $C \in dom(CT)$; (2) $CT(X) = \texttt{mixin}\ X\ \ldots$ for every $X \in dom(CT)$; (3)`Object` $\notin dom(CT)$; (4) $T \in dom(CT)$ for every class name and mixin name appearing in $ran(CT)$; (5) there are no cycles in the subtype relation induced by CT; (6) there are no field hidings of a class or a mixin by its subtype, whose subtyping relation is induced by CT.

In the induction hypothesis, we abbreviate $CT(C) = \texttt{class}\ C \ldots$ and $CT(X) = \texttt{mixin}\ X\ \ldots$ as $\texttt{class}\ C\ \ldots$ and $\texttt{mixin}\ X\ \ldots$, respectively.

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

The fields of type T, given in Figure 5, written $\mathit{fields}(T)$, is a sequence $\bar{T}\ \bar{f}$ pairing the type of each field with its name. If T is a class, $\mathit{fields}(T)$ is a sequence for all the fields declared in class T and all of its superclasses. 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 and a class. For the field lookup, we also have the definition

[4] We have solved the overloading problem that was faced by Jam.

$\dfrac{\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\bar{T}\ \bar{f};\ K_C\ \bar{M}\} \quad \mathit{fields}(\bar{X} :: D) = \bar{S}\ \bar{g}}{\mathit{fields}(C) = \bar{S}\ \bar{g},\ \bar{T}\ \bar{f}}$	$\mathit{fields}(\texttt{Object}) = \cdot$
	$\dfrac{\mathit{fields}(X) = \bar{T}\ \bar{f} \qquad \mathit{fields}(T) = \bar{S}\ \bar{g}}{\mathit{fields}(X :: T) = \bar{S}\ \bar{g},\ \bar{T}\ \bar{f}}$
$\dfrac{\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\bar{T}\ \bar{f};\ K_X\ \bar{M}\}}{\mathit{fields}(X) = \bar{T}\ \bar{f}}$	$\dfrac{\mathit{fields}(T) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, T) = T_i}$

Fig. 5. Field lookup

of $\mathit{ftype}(f_i, T)$ that is a type of field f_i declared in T. In contrast with McJava, field hiding is not allowed in Core McJava.

The 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 6 by S that is a result type. If T is a composition, the left operand of :: is searched first. If m with argument types $\bar{T}$ is not found in T, we define it `nil`. The type of method m in interface I is also defined in the same way. Similarly, the body of method m declared in type T with argument types $\bar{T}$, written $\mathit{mbody}(m, \bar{T}, T)$, is a pair, written $\bar{x}.e$ of a sequence of parameters $\bar{x}$ and an expression e. As mentioned earlier, in contrast with FJ, method overloading is allowed in Core McJava.

Typing. The typing rule for compositions is 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, 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. Well-formedness of class types and mixin types are straightforward and omited in this paper.

Figure 8 shows the typing rules for expressions. 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". These rules are syntax directed, with one rule for each form of expressions. Most of them are straightforward extension of the rules in FJ. The typing rules for constructor and method invocations check that the type of each argument is a subtype of the corresponding formal parameter. The typing rule for constructor invocation also assures that there are no instances of mixins and mixin-mixin compositions.

Figure 9 shows the typing rules for methods, classes and mixins. The type of the body of a method declaration is a subtype of the declared type, and, for a method in a class, the static type of the overriding method is the same as that of the overriden method. A class definition is well-formed if all the methods declared in that class and the constructor are well-formed. Similarly, a mixin is well-formed if all the methods declared in that mixin are well-formed.

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

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\bar{T}\ \bar{f};\ K_C\ \ \bar{M}\} \\ S\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{mtype(m, \bar{S}, C) = S}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\bar{T}\ \bar{f};\ K_C\ \ \bar{M}\} \\ S\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \notin \bar{M}\end{array}}{mtype(m, \bar{S}, C) = mtype(m, \bar{S}, \bar{X} :: D)}$$

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\bar{T}\ \bar{f};\ K_C\ \ \bar{M}\} \\ S\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\end{array}}{mbody(m, \bar{S}, C) = \bar{x}.e}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\bar{T}\ \bar{f};\ K_C\ \ \bar{M}\} \\ S\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \notin \bar{M}\end{array}}{mbody(m, \bar{S}, C) = mbody(m, \bar{S}, \bar{X} :: D)}$$

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

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

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

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

Fig. 6. Method lookup

Dynamic semantics. 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 $\longrightarrow$.

The reduction rules are given in Figure 10. There are two reduction rules, one for field access and one for method invocation. The field access reduces to the corresponding 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 (we write $[\bar{d}/\bar{x}, e/y]e_0$ for the result of substituting x_1 by d_1,...,x_n by d_n and y by e in e_0). Note that a

$$\frac{\begin{array}{c} \mathit{fields}(X) \cap \mathit{fields}(T) = \emptyset \\ \texttt{interface}\ I\ \{\bar{M}_I\} \quad (1) \quad (2) \\ \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ \ldots\ \bar{M}\ \} \end{array}}{X :: T \text{ ok}} \quad \text{(T-COMP)}$$

where

$$(1) = \begin{array}{l} \forall (S\ m(\bar{T}\ \bar{x})\{\ldots\}) \in \bar{M} \\ \mathit{mtype}(m, \bar{T}, X) = \mathit{mtype}(m, \bar{T}, T) \text{ or} \\ \mathit{mtype}(m, \bar{T}, T) = \texttt{nil} \end{array}$$

$$(2) = \begin{array}{l} \text{If } T \text{ is a composition } \bar{X} :: C, \text{ then} \\ \forall (U\ n(\bar{S}\ \bar{x})) \in \bar{M}_I \\ \mathit{mtype}(n, \bar{S}, I) = \mathit{mtype}(n, \bar{S}, T) \end{array}$$

Fig. 7. Well-formed composition

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

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

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

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

Fig. 8. Expression typing

$$\frac{\begin{array}{c} \bar{x} : \bar{T}, \texttt{this} : C \vdash e_0 : U_0 \qquad U_0\ <:\ T_0 \\ \texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\ldots\} \\ T_0 \text{ ok} \qquad \bar{T}\ \mathit{ok} \\ \text{if } \mathit{mtype}(m, \bar{T}, \bar{X} :: D) = S_0, \text{ then } S_0 = T_0 \end{array}}{T_0\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e_0\texttt{;}\ \}\ \texttt{OK IN}\ C} \quad \text{(T-CMETHOD)}$$

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

$$\frac{\begin{array}{c} K_C = C(\bar{S}\ \bar{g},\ \bar{T}\ \bar{f})\{\ \texttt{super(}\bar{g}\texttt{);}\ \texttt{this.}\bar{f}\texttt{=}\bar{f}\texttt{;}\} \\ \mathit{fields}(\bar{X} :: D) = \bar{S}\ \bar{g} \qquad \bar{M}\ \texttt{OK IN}\ C \\ \bar{X} :: D \text{ ok} \qquad \bar{T} \text{ ok} \end{array}}{\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\bar{T}\ \bar{f}\texttt{;}\ K_C\ \bar{M}\}\ \texttt{OK}} \quad \text{(T-CLASS)}$$

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

Fig. 9. Well-formed definitions

$$\frac{\mathit{fields}(\bar{X} :: C) = \bar{T}\ \bar{f}}{\texttt{new}\ \bar{X} :: C(\bar{e})\texttt{.f}_i \longrightarrow e_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(m, \bar{T}, \bar{X} :: C)\ =\ \bar{x}.e_0}{\texttt{new}\ \bar{X} :: C(\bar{e}).m\texttt{<}\bar{T}\texttt{>}(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, \texttt{new}\ \bar{X} :: C(\bar{e})/\texttt{this}]e_0} \quad \text{(R-INVK)}$$

Fig. 10. Operational semantics

method lookup in method invocation uses static types of arguments, using type annotations $\bar{T}$.

Properties. We show that Core McJava is type sound. The proof is given in the preliminary version of this paper [16]. [5] Intuitively, the step of proving Core McJava type soundness theorem is almost the same as that of FJ, but details vary a little.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T'$ for some $T'\ <:\ T$.*

[5] In this paper we omit type casts from [16] because they are less relevant to what we discuss in this paper.

```
class A {
  int f(M m) { ... }
  boolean f(M::C h) { ... } }
mixin M { // mixin M requires no interfaces
  void g() {
    int i = new A().f(this);
    ... }}
class Test {
  public static void main(String args[]) {
    new M::C().g(); }}
```

Fig. 11. An example program

Theorem 2 (Progress). *Suppose e is a well-typed expression.*

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

To state type soundness formally, we introduce a value v of an expression e by $v ::=$ `new` $\bar{X} :: C(\bar{e})$.

Theorem 3 (Core McJava 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* $U\ <:\ T$.

4 IMPLEMENTING MCJAVA

So far, we have overviewed the semantics of McJava. In this section, we show a compilation strategy from McJava programs to Java programs.

Outline of the compilation strategy. In order to explain the translation process, we start with a simple example code shown in Figure 11.

At the first step, the translator creates a file `A.java` from a class `A`. Then, it writes the body of class declaration into that file. At the beginning, the translator just copies the body of class `A` into `A.java`. Eventually, the translator encounters a composition type `M::C` that is not allowed in Java syntax. To compile this composition, the translator generates a new class `M_C` and replaces the occurrence of `M::C` with `M_C`. The class `M_C` extends a class `C` and implements an interface `M` that contains interface method declarations extracted from mixin `M` (Figure 12). The resulting class and interface are as follows:

```
interface M { void g(); }
class M_C extends C implements M {
  void g() { int i = new A().f((M)this); ... }}
```

Note that `this`, an argument of method invocation `f`, is type-casted to `M`. This casting is required, because in the translation `this` has type `M_C` that is

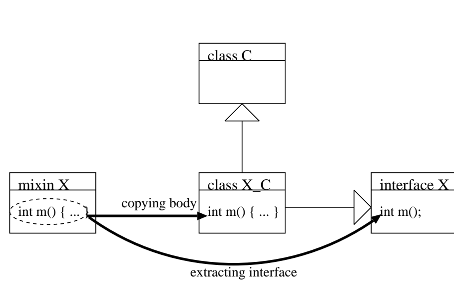

Fig. 12. Translation into Java classes

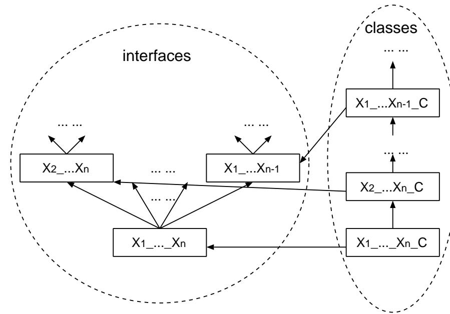

Fig. 13. Linearizing inheritance chain

subtype of both `M` and `C`, but `M` and `C` are not comparable. Without the type-cast, if class `C` has another method `String f(C m)`, the translated Java program will be ill-typed.

The translator also replaces the occurrence of `M::C` with `M_C` in the class `Test`:

```
class Test {
  public static void main(String args[]) {
    new M_C().g(); }}
```

So far, a simple case is explained, but McJava supports higher order mixins and flexible subtyping among them. We now describe a more general case. A composition $\mathtt{X}_1\mathtt{::}\cdots\mathtt{::X}_n$, where each $\mathtt{X}_i$ $(i \in 1 \cdots n)$ is a mixin, is translated into an interface $\mathtt{X}_1_\cdots_\mathtt{X}_n$ that extends all its immediate super types (shown by S-COMP in Figure 4). The body of this interface is empty. A composition $\mathtt{X}_1\mathtt{::}\cdots\mathtt{::X}_n\mathtt{::C}$, where each $\mathtt{X}_i$ $(\in 1 \cdots n)$ is a mixin and `C` is a class, is translated into a class $\mathtt{X}_1_\cdots_\mathtt{X}_n_\mathtt{C}$ that implements interface $\mathtt{X}_1_\cdots_\mathtt{X}_n$ and extends all its immediate super types other than $\mathtt{X}_1_\cdots_\mathtt{X}_n$ (i.e. its immediate super types whose the rightmost operands of `::` are `C`). Because a Java class can inherit only a single class, the class $\mathtt{X}_1_\cdots_\mathtt{X}_n_\mathtt{C}$ cannot extends so many classes at once; instead, they are linearized in a single inheritance chain (Figure 13)[6]. If class `C` has constructors, the default constructor of `C` is private. In this case, the McJava compiler writes constructor declarations that just invoke `super` in all the descendant classes of `C`.[7]

Optimizing compilation. The major problem of McJava compilation strategy explained above is, when we have a composition $X_1 :: \cdots :: X_n :: C$, then

[6] Making $\mathtt{X}_1_\mathtt{C}$ be a subclass of $\mathtt{X}_2_\mathtt{C}$ seems to be harmful, when an $\mathtt{X}_1_\mathtt{C}$'s method overrides an $\mathtt{X}_2_\mathtt{C}$'s method. To avoid this accidental overriding, the McJava compiler inserts a code that checks the type of `this` by using the `instanceof` operators, to make the appropriate body of method to be executed.

[7] Currently McJava forbids to declare a constructor for mixins.

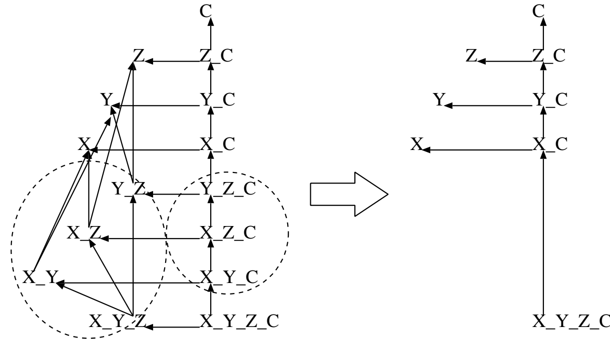

Fig. 14. Optimizing translation

we need to consider 2^n combinations that preclude a scalable compilation algorithm. Even though the situation where computing this combinations takes unacceptable ammount of time (e.g. the situation to compose more than 10 mixins at once) seems to be practically rare in the real problems, the introduction of an unnecessarily deep inheritance chain by linearization may results in critical run-time overhead. Indeed, the depth of inheritance becomes 2^n with the above algorithm.

Fortunately, type names that are not used in the source program can be removed from the inheritance chain. Assume the situation where we have a type `X::Y::Z::C`, but other compositions such as `X::Y::C`, `Y::Z::C` etc. are actually not used in the program (Figure 14). In this case, type names appear inside the dashed ovals can be removed from the inheritance chain, resulting in a new inheritance chain that is shown in the right hand side of Figure 14. By using this technique, almost all the overhead imposed by the linearization becomes acceptable.

The algorithm of optimization is explained below:

1. McJava compiler prepares a table $\mathcal{T}$ that contains all the type names used in the program. This process requires the whole program analysis.
2. Construct a connected graph *graph*(T) from a composition type T. For example, *graph*$(X :: Y :: Z :: C)$ becomes the left hand side of Figure 14.
3. For all the *class types* $\bar{X} :: C$ that are not in $\mathcal{T}$ except the class C and compositions that are composed with exactly one mixin, do the following operations:
   (a) Delete two edges $(U, \bar{X} :: C)$ and $(\bar{X} :: C, S)$, where both U and S are class types, from *graph*(T).
   (b) Insert a new edge (U, S) into *graph*(T).
4. For all the *mixin types* $\bar{X}$ that are not in $\mathcal{T}$, delete all the edges $(U_i, \bar{X})$ and $(\bar{X}, S_i)$ from *graph*(T).

Evaluating the translation. Current McJava compiler does not support separate compilation. This does not necessarily mean that current McJava compiler is impractical. Actually there are some practical systems that do not support separate compilation such as some C++ compilers [1] and AspectJ compiler [5]. It is clear, however, that support for separate compilation is very helpful to distribute binary form of mixins. For this purpose, we are thinking of introducing a *stub generator* that compiles mixins and a *linker* that composes the binary mixins before load time.[8]

McJava compiler is backward compatible to standard Java compilers.[9] That is, every Java program that can be compiled with a standard Java compiler can also be compiled with the McJava compiler.

Implementation status. At the moment we have developed a preliminary version of McJava compiler that has some restrictions including that it does not support access to Java standard libraries. The latest version of McJava compiler is downloadable at `http://kumiki.c.u-tokyo.ac.jp/~kamina/mcjava/`.

5 RELATED WORK

Jam [3] is a smooth extension of Java with mixins. Jam gives semantics of mixin compositions by translation to Java that is informally expressed by the *copy principle*. Even though this semantics looks natural, it has a serious problem in method overloading resolution, especially when an overloaded method is invoked with `this` as an argument. For example, a Jam program written the same as Figure 11 is not typable. This problem never occurs in McJava. Furthermore, due to the copy principle, it is very difficult to add higher order mixins in Jam.

Another approach of implementing a mixin is to parameterize a superclass of a generic class using a type parameter [23, 27, 22, 2]. One of the restrictions of McJava that is not shared by the generic type approach is its disability to access mixin's superclass type inside the mixin, e.g.:

```
class Color<Widget extends WidgetI> extends Widget {
  Widget f;
  ... }
```

We may partially solve this problem by adopting a coding convention to make the classes composed with the mixin explicitly implement the constraint (the required interface) of that mixin. Another possible design of McJava is to impose a superclass of the mixin to explicitly implement the required interface. In other words, the superclass must be a subtype of the required interface. However, there is a tradeoff. The reason why we take the approach of structural constraint, where a superclass of mixin must be a *structural* subtype of required interface, is that it is more flexible for compositions. Mixins are often implemented *after* the

[8] The idea is taken from Jiazzi [19].

[9] Except that McJava reserves keywords `mixin` and `requires`.

implementation of possible superclasses. Imposing these classes to be a nominal subtype of required interface is rather restrictive, because it would require re-implementation of the original classes. Another difference between generic classes and McJava is the flexibility of subtyping. Generic classes cannot capture the full power of McJava type system, where a mixin may be used as a type, and `Color::Font` is a subtype of both `Color` and `Font`.

Besides the feature of structural constraints on mixin's superclasses, McJava is a *nominally typed* class-based language, that means the name of a class (or mixin) determines its subtype relationship. On the other hand, in object-oriented languages with *structural subtyping*, the subtype relation between classes is determined by their structures. A core calculus of classes and mixins for structurally typed language was proposed by Bono et al.[6]. Instead, we take a nominal approach, because the target language (Java) is nominally typed.

To our knowledge, a core calculus for mixin types extending Java was originally developed by Flatt et al.[13]. The novel feature of this calculus, named MixedJava, is its ability to support hygienic mixins [2, 19]. Hygienic mixins use the static type information when looking up a method, avoiding the problem of method collision. This feature is achieved by changing the protocol of method lookup: in MixedJava, each reference to an object is bundled with its *view* of the object, the run-time context information. A view is represented as a chain of mixins for the object's instantiation type. It designates a specific point in the full mixin chain, the static type of that object, for selecting methods during dynamic dispatch. Even though the proposal of hygienic mixins is useful, there is no implementation of MixedJava. However, there exists two kinds of implementation of hygienic mixins [2, 19], each of them does not conform the McJava type system either; without support for hygienic mixins, McJava defines very flexible subtyping relations. For example, the subtype relation $X :: Y :: C \ <: \ X :: C$ is missing in MixedJava. Our work of adapting the implementation strategies of hygienic mixins cited above to our McJava compiler is in progress, and the result will be published in elsewhere.

Mixin modules [10], essentially motivated by the problem of interaction with recursive constructs that cross module boundaries in module systems of functional languages, mainly focus on facilitating reusing large scale programming constructs such as frameworks [11]. Our work, on the other hand, mainly focuses on integrating mixin-types and its flexible subtyping with real programming languages. The work [11] sacrifices mixin subtyping in favor of allowing method renaming. MixJuice [14] is also independently proposed by Ichisugi et al. to modularize large scale compilation unit.

Schärli et al. proposed *traits* [21], fine grained reusable components as building blocks for classes. Traits support method renaming that overcomes the problem of method collision. When traits are composed, the members of those traits are "flattened" into one class, which also solves the ordering problem of mixins. Our work, in contrast with traits, has more focus on declaring a mixin as a type, and studying their subtype relations. We also would like to note that the ordering of mixins is useful particularly when we "extend" a parametrized superclass

with the same name of method as the superclass, and invoke it via `super.m`, where *m* is a method name.

Mixins may be used as vehicles to directly implement *roles* in terms of role modeling [24]. Epsilon [26, 25], a role-based executable model, was also proposed for this purpose. While Epsilon has a feature of dynamic object adaptation, we consider McJava and its core calculus provides a bood basis for incorporating static typing into Epsilon.

6 CONCLUSIONS

This paper reports the design and implementation of programming language McJava that is an extension of Java with mixin-types. In McJava, a mixin is a type, and a composition is a subtype of its constituents. McJava's semantics is different from Jam's copy principle; we solved a bothersome problem of using `this` inside mixins Jam faced, and provide a strong feature of higher order mixins and flexible subtyping. We formally present a type system and operational semantics of Core McJava, a small calculus of McJava, that gives an assurance that McJava type system is sound.

Acknowledgements: The authors would like to thank Atsushi Igarashi, Hidehiko Masuhara and Etsuya Shibayama for their very helpful comments on the earlier version of this paper. 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. GCC home page. http://gcc.gnu.org/.
2. Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proceedings of OOPSLA2003*, pages 96–114, 2003.
3. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – A smooth extension of java with mixins. In *ECOOP 2000*, pages 154–178, 2000.
4. Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
5. AspectJ. http://www.eclipse.org/aspectj/.
6. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A Core Calculus of Classes and Mixins. In *Proceedings of ECOOP'99*, LNCS 1628, pages 43–66, 1999.
7. Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.
8. Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.
9. Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, 1992.
10. Dominic Duggan and Constantinous Sourelis. Mixin modules. In *ICFP'96*, pages 262–272, 1996.

11. Dominic Duggan and Ching-Ching Techaubol. Modular mixin-based inheritance for application frameworks. In *OOPSLA 2001*, pages 223–240, 2001.
12. Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of ICFP 1998*, pages 98–104, 1998.
13. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL 98*, pages 171–183, 1998.
14. Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism. In *Proceedings of ECOOP 2002*, pages 62–88, 2002.
15. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
16. 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`.
17. Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Arts of the Metaobject Protocol*. The MIT Press, 1991.
18. 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.
19. Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of OOPSLA2001*, pages 211–222, 2001.
20. D. A. Moon. Object-oriented programming with flavors. In *OOPSLA'86 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 1–8, 1986.
21. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP 2003*, LNCS 2743, pages 248–274, 2003.
22. Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings ECOOP'98*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570, 1998.
23. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
24. Tetsuo Tamai. Objects and roles: modeling based on the dualistic view. *Information and Software Technology*, 41(14):1005–1010, 1999.
25. 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.
26. 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 *Lecture Notes in Computer Science*, pages 89–109, 2001.
27. Michael VanHislt and David Notkin. Using C++ templates to implement role-based designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.