平成16年度博士学位論文

A Design and Implementation of Mixin-Based Composition in Strongly Typed Object-Oriented Languages

強く型付けされたオブジェクト指向言語におけるmixin合成の設計と実現

The University of Tokyo
Graduate School of Arts and Sciences
Department of General Systems Studies

東京大学大学院 総合文化研究科
広域科学専攻

Tetsuo Kamina
紙名 哲生

Contents

1 Introduction **7**

1.1 Background . . . 7

1.1.1 Recent Tendency of Language Design . . . 7

1.1.2 Problems in Single Inheritance of Java . . . 8

1.1.3 Mixins in Flavors and CLOS . . . 10

1.2 Subject of the Dissertation . . . 11

1.3 Our Contributions . . . 12

1.4 Organization of the Dissertation . . . 14

2 McJava: Designing Java with Mixin-Based Composition **15**

2.1 Mixin Declarations and Mixin-Types . . . 15

2.2 Higher Order Mixins . . . 18

2.3 Mixin-Based Subtyping . . . 18

2.4 Mixin Composability . . . 21

2.5 Current Limitations . . . 21

2.6 Case Study: Integrated Systems . . . 23

2.7 Summary . . . 28

3 Core McJava: A Core Calculus of McJava **29**

3.1 Syntax . . . 30

3.2 Class Table . . . 31

3.3 Auxiliary functions . . . 32

3.4 Typing . . . 33

3.5 Dynamic Semantics . . . 37

3.6 Properties . . . 38

3.7 Summary . . . 43

4 Implementation of McJava 45

4.1 Outline of the Compilation Strategy 45
4.2 Evaluating the compilation 50
4.3 A Sketch for Separate Compilation 52
4.4 Summary 52

5 An Advanced Mechanism of Method Dispatch 53

5.1 The Problem of Accidental Overriding 53
5.2 Selective Method Combination 57
5.3 Implementation Issues 60
5.4 Summary 63

6 Mixins and Other Language Features 65

6.1 Generics 66
6.2 `ThisType` 69
6.3 An Approach to Layered Design 71
6.3.1 Mixin Layers 71
6.3.2 Our Approach to Generic Mixin Layers 73
6.4 Discussion 78
6.5 Summary 79

7 Related Work 81

7.1 Mixin-Based Systems 81
7.1.1 Jam: Another Approach to Java with Mixins 81
7.1.2 Other Mixin-Related Systems 83
7.2 Method Combination in Object-Oriented Languages 86
7.3 Other Related Issues 87

8 Conclusion 89

8.1 Summary of the Dissertation 89
8.2 Future Work 90

List of Figures

1.1 Subtyping anomaly in Java 9

2.1 A color mixin 16
2.2 Label and text field classes 16
2.3 A Font mixin 19
2.4 An integrated system 24
2.5 Equality in McJava 25
2.6 A role for equality in McJava 26
2.7 An example program of integrated systems 27

3.1 Abstract syntax of Core McJava 30
3.2 Subtype relation 32
3.3 Field lookup 33
3.4 Method type lookup 34
3.5 Method lookup 35
3.6 Well-formed composition 36
3.7 Expression typing 37
3.8 Well-formed definitions 38
3.9 Operational semantics 39

4.1 Translation into Java classes 46
4.2 An example program illustrating the compilation 47
4.3 Translation into Java classes (a complex case) 50

5.1 Accidental Overriding in McJava 54
5.2 New method lookup in McJava 58
5.3 Compiled code of Figure 5.1 and `Id` (1) 61
5.4 Compiled code of Figure 5.1 and `Id` (2) 62

6.1 An example of generic class 66
6.2 Composition of type parameters 68
6.3 An example of error using `ThisType` 70
6.4 Layered Design 71
6.5 Mixin layers implementation using C++ 72
6.6 A generic graph layer 74
6.7 A generic graph layer 75
6.8 A dense graph module 76
6.9 A generic traversal layer 77
6.10 A depth first visitor module 77

7.1 An example of faulty Jam code 82

Acknowledgments

I have many people to thank for helping me complete my thesis. First of all, I cordially thank Tetsuo Tamai, my thesis advisor, who has advised me how to plan a good research, how to proceed the research effectively, and how to write impressive papers. He helped me learn how to study both on the theoretical and practical point of views.

I express my appreciation to all the members of Prof. Tamai's laboratory and PPP research group for their insightful comments on my work. Especially, Hidehiko Masuhara has given me many helpful comments on language design and efficient implementation. He has also helped me learn how to make effective oral presentations.

I would like to thank all the members of the Kumiki project for fruitful discussion at the monthly meetings. Especially, Atsushi Igarashi has given me many suggestions on the McJava type system. The subtyping like `X::Y::Z <: X::Z` is actually owed to his suggestion. Etsuya Shibayama and Shin Nakajima gave me many helpful comments on my research while the discussion.

I also express my appreciation to all the members of KTYY research group for giving me the opportunities of oral presentation and fruitful discussion, especially to insightful comments from Tetsuro Tanaka and Tomoyuki Kaneko.

I would like to thank all my friends who have been keeping me sane during my graduate student years, which can easily be discouraging and stressful.

Finally, I express special thanks to my parents, Yoichi Kamina and Takako Kamina, for everything they have done for me over the years. Without their encouragement, sustaining help, and toleration, I could not finish my thesis.

December 20, 2004
Tetsuo Kamina

Chapter 1

Introduction

1.1 Background

1.1.1 Recent Tendency of Language Design

In the past years, simple object-oriented languages were considered more preferable to programmers than more powerful but complex languages, because complex languages sometimes become error prone. For example, the complexity of multiple inheritance was criticized because it produces more problems than what it solves (e.g. violation of encapsulation in CLOS by linearization of multiple superclasses) [55]. In many object-oriented languages such as Smalltalk, Java, and C#, code reuse is supported by simpler mechanism of single inheritance and overriding; using them, programmers may derive a new class by specifying only the elements to be extended and modified from the original class. Actually, the type system of Java, one of the most widely-used strongly typed object-oriented languages, can be characterized by simple constructs such as classes, single inheritance and subtyping, method and field declarations, and so on [34].

Today, however, this tendency seems to be over. Simple languages have turned out to be less expressive than what programmers really want. There are many evidences that more powerful *and* safe languages are considered more desirable. For example, the recent version of Java offers full expressive power of parametric polymorphism [11] and wildcards [64, 36]. Furthermore, current enthusiasm on aspect-oriented programming [39] implies that many programmers think the current object-oriented languages are less expressive for modularizing program pieces. Extensive efforts are now being taken to extend Java to more expressive languages [20, 33, 38, 3, 61, 17, 14, 31, 28].

This dissertation stems from the above observations. In particular, the widely-used single inheritance mechanism lacks an ability of reusing uniform extensions and modifications to multiple classes.

1.1.2 Problems in Single Inheritance of Java

Java-like languages have a problem in reusability. To see this problem, suppose we are developing a widget library for GUI components. For example, we will declare a class `Label`. Then, we will extend it to get a subclass with the "color" feature such as an attribute that represents the color of the label, a method to change the color, and so on. We will also declare a class `Text`, and extend it to get a subclass of `Text` with the "color" feature in the same way. In this case, the code for the "color" feature is duplicated in multiple classes thus degrading maintainability of the program.

One way to avoid such duplication is to use *design patterns* [30] such as the *decorator pattern*. However, design patterns impose an additional workload on programmers; e.g., we have to invent some additional classes that are not essential in the problem domain. Furthermore, discovering which pattern is used in the program is rather difficult, because patterns are implicit; i.e., there are no language constructs for describing patterns, making it difficult to understand the program.

The problem is that there are no straightforward way for modularizing the "color" feature. In fact, Java does not provide a construct to modularize such *uniform extensions and modifications* to multiple classes.

Another problem of Java is found in subtyping. In Java, we can explicitly denote a class is a subtype of another class or interface by using the `extends` or `implements` clause[1]. Furthermore, this subtype relation is transitive; e.g., a class is a subtype of its immediate superclass's superclass. However, there are no subtype relations between classes that extend the same class through different inheritance paths, even when they implement the same

[1]We can also denote a interface is other interfaces

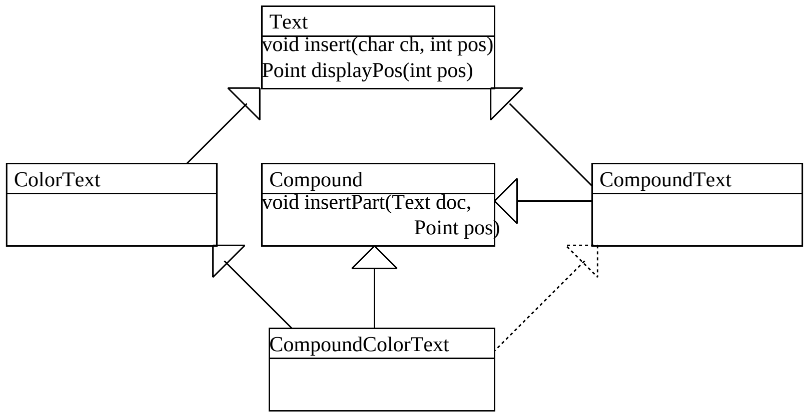

Figure 1.1: Subtyping anomaly in Java

interface. Figure 1.1 illustrates this problem. The class `CompoundText` extends `Text` and implements the interface `Compound`. `ColorText` extends `Text`. The class `CompoundColorText` extends `ColorText` and implements the interface `Compound`. In this case, we should be able to use an instance of `CompoundColorText` in the context of `CompoundText`, because the former class implements all the features provided by the latter class. However, Java does not allow this usage, because there are no subtype relations between `CompoundColorText` and `CompoundText`. Therefore, even though execution of the following code will be safely performed, the Java compiler reports an error.

```
class LibraryServices {
  void insertDocPart(Text doc, CompoundText ct, int pos) {
    ct.insertPart(doc, ct.displayPos(pos));
    ...
  }
}

CompoundColorText cct = new CompoundColorText();
LibraryServices ls = new LibraryServices();
...
ls.insertDocPart(doc, ct, pos) // Compile error!!
```

This inflexibility in subtyping should be avoided in some ways.

1.1.3 Mixins in Flavors and CLOS

A programming construct *mixin* (also known as *abstract subclass*) was invented in object-oriented extensions of Common Lisp such as Flavors [43] and popularized by CLOS [37]. By using mixins, we can implement uniform extensions and modifications to classes.

A mixin is a partially implemented subclass whose superclass is not provided in its declaration; we may provide a variety of actual classes to the mixin to create a concrete class. For example, in CLOS we may declare a mixin `Color` with the declaration:

```
(defclass Color () (color))
(defmethod paint ((self Color) (g Graphics))
  (setColor g (slot-value self 'color))
  (call-next-method g))
```

The `defclass` construct includes the name of the new class, a list of its superclasses, and a list of its instance variables (a list of slots in CLOS terminology). The argument list of the `defmethod` construct declares the class on which the method is defined. The expression `call-next-method` plays the role of `super` in Java.

In the above definition, `Color` does not declare any superclasses, but it invokes `call-next-method`. This invocation obviously leads to an error unless a superclass of `Color` is provided. The CLOS *linearization* mechanism plays an important role for providing it; i.e., we can declare a subclass of both `Color` and its "superclass", e.g., `Label`:

```
(defclass ColorLabel (Color Label) ())
```

In CLOS, if a class inherits from multiple classes, and these classes declare methods with the same signature, these methods are combined by `call-next-method`. Which method is executed by the call of `call-next-method` is determined by the order of the list of superclasses declared in `defclass`. In the above case, the execution of `paint` defined in `Color` always precedes `paint` defined in `Label`. Therefore, we can combine the `paint` methods declared on `Color` and `Label`.

The class `Color` is a mixin that provides uniform extension and modification to multiple widget classes. We may compose it with classes other than `Label`:

```
(defclass ColorText (Color Text) ())
```

Note that the mixin in CLOS is simply a coding convention and has no formal status. In CLOS, every class can be a mixin if it uses `call-next-method` that is not bound to any superclasses, whereas in our approach explained below, mixins are explicitly declared with a new syntactical form.

Mixins provide 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 has been studied both on the methodological and theoretical point of views [9, 10, 5, 8, 27]. Small core languages that support mixins or *mixin modules* are also proposed [29, 23]. Despite the existence of these extensive studies, relatively few attempts are made on designing *real* strongly typed programming languages that support mixins[2].

1.2 Subject of the Dissertation

This dissertation addresses how to design and implement the mixin mechanism in nominally typed object-oriented languages like Java to solve the aforementioned problems. There are many technical problems in the language design and implementation treated in this dissertation:

Language Design and Type Soundness. Type soundness is one of the most basic properties of programming languages that ensures well-typed programs do not "go wrong." Most modern programming languages such as Java are designed to hold this property. To design an extension of such a language, we should carefully study how the new constructs interact with the existing constructs to ensure that the new constructs do not behave unsafely. In other words, we should *prove* that the property of type soundness still holds in the language extending Java with mixins.

[2]There are some work related to our approach such as Jam [4] that integrates mixins with Java. We will note differences between these work and our approach in Chapter 7.

Implementation. New language constructs should not degrade run-time performance of the original language. Furthermore, if we consider compatibility with the existing run-time systems, the new constructs should also run efficiently on the current standard platforms. This implies that we should carefully study compilation strategy of mixins.

Assurance of Behavioral Consistency. Sometimes, a new programming language construct, which solves some problems, also produces new problems, even when it holds that the type system is sound. One problem that mixins raise is known as *accidental overriding*. This problem stems from the fact that an implementor of a mixin does not know what superclass the mixin will be composed with. Therefore, when a user of a mixin (who will be different from the implementor of that mixin) tries to compose it with some other classes, it is possible that a method declared in the mixin accidentally overrides a method declared in the superclass. This overriding is harmful because it accidentally changes the behavior of the superclass, so it should be avoided in some ways.

Interaction with Other Constructs. As mentioned before, there are extensive efforts in extending Java with other useful constructs that are originally not related to mixins. For example, there are many efforts on adding parametric polymorphism to Java [47, 11, 1, 19] (one of which, [11], is actually included in the official release of Java). Another example is introducing the type of self (`MyType`) [13] to Java [14] (in this context, it is called `ThisClass` or `ThisType`). It is interesting to study how these advanced issues interact with mixins.

1.3 Our Contributions

The contributions of this dissertation are summarized as follows:

- We design a programming language McJava[3], an extension of Java with mixins. McJava provides a new syntactic form for explicitly declaring mixins. A mixin can be composed with a variety of superclasses by using a composition operator. Furthermore, McJava supports more advanced

[3] **M**ixin-based **C**ompositions for **Java**

features such as *higher order mixins* [6] that is a mechanism for allowing a mixin to be composed with another mixin, resulting a new mixin, and mixin-based subtyping, a flexible subtyping relation defined among mixin compositions.

- We develop Core McJava, a small subset of McJava that offers a few key constructs that characterize the type system of McJava. We then develop a proof of type soundness of Core McJava. As a vehicle for our study of Core McJava, we extend Featherweight Java [34], a small core language of Java.

- We implement a McJava compiler that compiles McJava programs into Java programs. This makes an assurance that McJava programs are runnable on any standard Java virtual machines. It also ensures that McJava do not degrade run-time performance of Java.

- To tackle the problem of accidental overriding, we equip a new method dispatch mechanism on McJava. This mechanism allows multiple methods with the same signature coexist on the same object; when a method is called on the object, the most specific method (that corresponds to the Java method dispatching rule on overriding) *from the viewpoint of the statically known type of the object* is selected to execute. McJava raises another non-trivial issue on this dispatching. Owing to its flexible mixin-based subtyping rules in McJava, an immediate superclass of a mixin in the run-time inheritance chain may be different from the statically known superclass, thus requiring a sophisticated treatment in invoking a superclass's method. We implement this method dispatching mechanism onto the McJava compiler.

- We study how mixins interact with generics and `ThisType` [14]. We design an extension of McJava with generics and `ThisType`, and informally discuss that the language is not type-sound, but we can recover type soundness of the language by imposing restriction on covariant subtyping among inner mixins. We also show how expressive the language is for code reuse by giving an example.

1.4 Organization of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we overview the design of McJava. After explaining each notable feature of McJava, we show an example that illustrates the usefulness of McJava. In Chapter 3, we develop Core McJava, a core calculus of McJava, that provides assurance on the soundness of McJava type system. After defining Core McJava with type derivations and reduction semantics, we show the type soundness theorem. Then, in Chapter 4, we present how to compile McJava programs to Java programs. Some non-trivial issues on implementation are also presented. In Chapter 5, we develop a new approach to method dispatch on mixin-based composition, which solves the problem of accidental overriding. This mechanism is called *selective method combination*. We show how this mechanism works appropriately with the presence of flexible mixin-based subtyping rules on McJava. In Chapter 6, we discuss how mixins interact with other language constructs such as generics and `ThisType`. We also show how the language (McJava with generics and `ThisType`) has expressive power for code reuse by using a graph traversal example. In Chapter 7, we discuss the relationship between our approach and other related work. Finally, in Chapter 8, we conclude this dissertation with further research directions.

Chapter 2

McJava: Designing Java with Mixin-Based Composition

This chapter overviews a programming language McJava, an extension of Java with mixins. McJava immigrates mixins from the context of dynamically typed languages such as Flavors [43] and CLOS [37] into a statically typed language. In McJava, a mixin is explicitly declared with a name, and the name of mixin can be used as a type (*mixin-types*). Furthermore, McJava supports more advanced features of mixins such as *higher order mixins* [6] and mixin-based subtyping.

2.1 Mixin Declarations and Mixin-Types

To demonstrate how a mixin is declared in McJava, we start with a very simple example. Figure 2.1 shows a declaration of mixin `Color`. This mixin provides the "color" feature that is intended to be composed with widget classes.

A statement beginning with keyword `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, both class `Label` and class `Text`, shown in Figure 2.2, can be composed with mixin `Color`, as they implement interface `WidgetI`.

```
interface WidgetI { void display(Graphics g); }

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

Figure 2.1: A color mixin

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

class Text {
  void display(Graphics g) { ... }
}
```

Figure 2.2: Label and text field classes

It is not necessary for these classes to explicitly declare that they implement interface `WidgetI`, as shown by the class `Text`. A class that implicitly implements a `display` method (i.e. a class that has a `void display(Graphics g)` method without declaring `implements WidgetI`) may also be composed with mixin `Color`, even though declaring the required interface explicitly helps programmers to understand the program.

Note that the `requires` clause of mixin declarations is quite different from the `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`, because `Color` need not implement `WidgetI`; it is the `Color`'s superclass's responsibility to implement the interface. The `requires` clause enables separate type-checking of mixins. In other words, 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, we can safely invoke `super.display(g)` in the `display` method of mixin `Color`, that results in invocation of `display` declared in `Color`'s "superclass".

The advantage of writing required interfaces separately is that they can be reused in other mixins. For example, interface `WidgetI` may be reused in other mixin, namely `Font`. However, writing required interfaces separately imposes programmers more workload. McJava therefore allows an anonymous interface to appear in `requires` clause for more handy syntax:

```
mixin Color requires {
  void display(Graphics g); } {
  ...
}
```

If a mixin requires *no* interfaces (i.e. a mixin that can be composed with any classes), we may omit the `requires` clause.

A composition of mixin `Color` and class `Label` is written as `Color::Label`. This composition is considered as a subclass that is derived from the superclass `Label`, with subclass body declarations being the same as the body of mixin `Color`. Similarly, composition `Color::Text` is considered as a subclass of `Text`. In this sense, a mixin is a uniform extension of classes that may be applied to many different superclasses. Because of this uniformness, we may not declare a superclass for a mixin by using an `extends` clause.

Besides this modularity, McJava also provides the useful feature of mixin-types, a mixin declared is also used as a type. Therefore, 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.

In McJava, it is forbidden to create an instance of a mixin, because an expression like:

```
new Color().display(g)
```

will result in invoking an unknown method. The design decision is natural, since a mixin is also known as an *abstract subclass*. Just as an ordinary abstract class cannot be instantiated in Java, creating an instance of mixin is not meaningful.

2.2 Higher Order Mixins

We have seen composition of a mixin and a class. A composition of a mixin and a class is considered to be a class; e.g., we can instantiate a composition by using `new` expression like `new Color::Label()`, or we can declare a new class that inherits from the composition.

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 2.3. This composition, written as `Color::Font`, is quite different from a composition of a mixin and a class; it is regarded as a mixin that has both features of `Color` and `Font`. This mechanism is called *higher order mixins*, in that a mixin can be an argument of composition, resulting a new mixin.

As shown before, a mixin declaration does not have an `extends` clause, because the superclass of a mixin is not provided in the declaration of the mixin. However, it is sometimes useful to extend and modify a mixin to refine the definition of the original mixin. The feature of higher order mixins compensates this ability.

Note that, in the case of higher order mixins, the right hand side mixin of composition operator `::` need not implement the required interface of left hand side mixin. The requirements of well-formed composition is informally illustrated in section 2.4 and formally discussed in Chapter 3.

2.3 Mixin-Based Subtyping

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

```
interface WidgetI { void display(Graphics g); }

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

Figure 2.3: A Font mixin

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 `display` 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* 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. With this restriction, it becomes easier to satisfy the Liskov's behavioral subtyping [40].

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 to respect 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.

One may wonder what happens when we compose the same mixins like `X::X`. Actually, this composition is allowed in McJava and that is a subtype of `X`. Even when methods declared in the right hand side `X` are overridden by their corresponding methods declaration in the left hand side, we may invoke the method on the right hand side mixin if the method is declared in the required interface and called by `super`. For example, if the following method

```
int f() { return super.f() * super.f(); }
```

is declared in `X`, invoking `X::X::C.f()` (where `C` is a class) results in invoking `C.f()` four times.

The subtyping rule proposed here solves the problem of subtyping in Java discussed in section 1.1.2. Suppose we have a mixin `Compound`:

```
mixin Compound {
  void insertPart(Text doc, Point pos) { ... }
  ... }
```

Since `Compound::Color::Text` is a subtype of `Compound::Text`, the McJava type system accepts the following code:

```
class LibraryServices {
  void insertDocPart(Text doc, Compound::Text ct, int pos) {
    ct.insertPart(doc, ct.displayPos(pos));
    ...
  }
}
```

```
Compound::Color::Text cct = new Compound::Color::Text();
LibraryServices ls = new LibraryServices();
...
ls.insertDocPart(doc, ct, pos)
```

With this subtyping, an immediate superclass of a mixin in the run-time inheritance chain is no longer necessarily the same as the statically known superclass. This fact raises another issue when we try to solve the problem of accidental overriding. It will be explained in Chapter 5.

2.4 Mixin Composability

Adding mixin-types to Java type system requires the type-checker to perform additional 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 and the same formal parameter types as *m* is also declared in `T`, then the return type of *m* must be the same as the type of *m'*.

The first rule is for composition of mixins and a class that can be instantiated. It 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` "overrides" a method declared in `T` with the different return type, the compiler reports an error.

2.5 Current Limitations

Current McJava has the following limitations.

Constructors. McJava forbids to declare a constructor for mixins. Although this restriction seems to lower usability, we believe that actually it does not. We may use a coding convention to declare an initializer `void initM(...)` (where `M` is a mixin name) that is responsible for initializing the instance variables of mixin `M`.

We impose this restriction to mixins because the role of a constructor is an instance generator but a mixin cannot be instantiated. A constructor should have responsibility for initializing not only instance variables of a mixin in which the constructor is declared but also instance variables of all the super classes. However, a mixin has no way to know the signature of super class's constructors.

Actually, constructors should not exist in mixins but in compositions to be instantiated. Indeed, Jam [4] takes this approach. Although there are no way for declaring members and constructors on compositions in McJava, we may obtain the same effect by declaring a new class whose superclass is a composition:

```
class H extends M::C {
  H(...) { super(...); this.initM(...); }
}
```

Static Members in Mixins. McJava forbids to declare static members inside mixins. There are some conceivable choices to define semantics of static members in mixins. One approach is that a member declared with `static` modifier inside a mixin is not considered as a static member of the mixin, but of its compositions. Jam takes this approach because it conforms to *copy principle* (explained in Chapter 7) well. Another approach is to share only one copy of the static members among all the compositions of that mixin. Currently we postpone the decision which approach to take.

Field Members in Mixins. Mixins are allowed to declare field members, but all fields must be declared as `private`. This restriction is due to the field hiding problem that is also discussed in Jam. Jam takes another approach that allows declaration of public field members. There is a design tradeoff between these two approaches. McJava may also take Jam's approach, although a little further research is required to ensure that allowing public field members in mixins is also sound, because the core language of McJava (explained in Chapter 3) does not allow field hiding.

2.6 Case Study: Integrated Systems

To explain the expressive power of McJava, we show an interesting example of integrated systems as a case study. In [58], an integrated system is defined as "a collection of software tools that work together, freeing the user from having to coordinate them manually." For example, an integrated system with tools for text editing, compiling, and debugging will ensure that when the debugger reaches a breakpoint, the editor scrolls to the corresponding source statement.

One of the main problems in implementing an integrated system is its difficulty for evolution. Managing the complexity of integrated systems is hard. The solution of this problem is separating the components (i.e. the integrated software tools) and their relations at the design and implementation levels; however, Sullivan et al. argued that an integrated system implemented by a conventional object-oriented language and even by an aspect-oriented language like AspectJ [38] hardly evolves [57][1]. In this section, we propose a solution to this problem with McJava, and show how the mechanism of mixin-types is used in this solution. This solution is partial, because it assumes that components and their relations are statically known. However, this example well describes how the mechanism of mixin-types supports modular construction of program pieces.

We show a simplified example of integrated systems originally described in [57]. In this example, the software tools that are subject to integration are binary objects that have two states, *on* and *off*. We call these objects *Bits*. An instance of Bit has operations named *set* and *clear*, to change its state to "on" and "off," respectively. Binary relations, *Equality* and *Trigger*, are defined between Bits. The Equality relation always makes the states of the related Bits the same, while the Trigger relation activates the *target* Bit to be "on" if the *source* Bit becomes "on," but takes no action on the other situations.

For example, let us assume the structure in Figure 2.4. In this system, the

[1]Enhancements of AspectJ that can solve this problem are also proposed [52, 50].

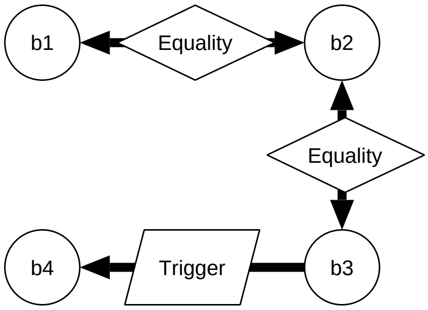

Figure 2.4: An integrated system

four objects, b1, b2, b3 and b4, are instances of Bit; b1 and b2, and b2 and b3 are connected by Equality relations; b3 is a trigger of b4. If b1 receives a message "set," then the "set" message is sent to b2, that also activates sending of the "set" message to b3. Furthermore, the "set" message is sent to b4, because b3 is a trigger of b4. However, no matter what is sent to b4, nothing happens to b3.

The problem is to make this system evolvable, separating the implementation of the Bit objects from the Equality and Trigger relations, and make this system modular and scalable. Modularity means implementation of relations should be able to adapt to other implementations of Bit objects, and the implementation of the Bit objects should be reusable in other contexts. Scalability means that we may add new Bit objects and even new relations other than Equality or Trigger to that system with no difficulty.

Figure 2.5 gives an implementation of `Equality` relation. An `Equality` is a binary relation, so it has two instance variables `role1` and `role2` to hold the Bit objects that are linked with the `Equality` relation. But we would like to apply this `Equality` to other implementations of Bit objects. Therefore, the type of `role1` and `role2` is declared as `EqAdaptor` that abstracts a set of operations the `Equality` is interested in.

`EqAdaptor` is declared as a mixin in Figure 2.6. It declares methods `set()` and `clear()`. Since those methods invoke `super.set()` and `super.clear()` respectively, `EqAdaptor` requires the interface `eqI` that declares `set()` and `clear()`. `EqAdaptor` may be composed with any class that implements the methods declared in `eqI`. For example, the following class `Bit` may be composed

```
class Equality {
  public boolean busy;
  EqAdaptor role1, role2;

  public void join1(EqAdaptor e) {
    role1=e;
    e.equalities.add(this);
  }
  public void join2(EqAdaptor e) {
    role2=e;
    e.equalities.add(this);
  }
  public EqAdaptor getOpponent(EqAdaptor e){
    if (role1 == e) return role2;
    else if (role2 == e) return role1;
    else return null;
  }
}
```

Figure 2.5: Equality in McJava

with `EqAdaptor`.

```
class Bit {
  boolean state=false;
  void set() { state=true; }
  void clear() { state=false; }
  boolean get() { return state; }
}
```

At first, the method `set()`/`clear()` of `EqAdaptor` invokes the corresponding method declared in the superclass (for example, the `set()`/`clear()` of `Bit` class). Then, it sends the `set()`/`clear()` message to all the objects that have the `Equality` relation linkage with the sender. The instance variable `busy` declared in `Equality` is a flag that ensures the transition of these method invocations does not end up with an infinite loop.

```
interface eqI {
  void set();
  void clear();
}

mixin EqAdaptor requires eqI {
  public Vector equalities = new Vector();

  public void set() {
    super.set();
    for (Iterator i=equailties.iterator();
         i.hasNext(); ) {
      Equality e = (Equality)i.next();
      if (!e.busy) {
        e.busy = true;
        e.getOpponent(this).set();
        e.busy = false; }}}

  public void clear() {
    super.clear();
    for (Iterator i=equalities.iterator();
         i.hasNext(); ) {
      Equality e = (Equality)i.next();
      if (!e.busy) {
        e.busy = true;
        e.getOpponent(this).clear();
        e.busy = false; }}}
}
```

Figure 2.6: A role for equality in McJava

```
class Main {
  public static void main(String[] args) {
    EqAdaptor::Bit b1=new EqAdaptor::Bit();
    EqAdaptor::Bit b2=new EqAdaptor::Bit();
    TrAdaptor::EqAdaptor::Bit b3 =
        new TrAdaptor::EqAdaptor::Bit();
    Bit b4 = new Bit();
    Equality e1 = new Equality();
    Equality e2 = new Equality();
    Trigger t1 = new Trigger();

    e1.join1(b1); e1.join2(b2);
    e2.join1(b2); e2.join2(b3);
    t1.join1(b3); t1.join2(b4);
    ...
  }
}
```

Figure 2.7: An example program of integrated systems

The Trigger relation is also implemented in the same way. Then, the integrated system may be implemented as in Figure 2.7. Because b1 and b2 only join in the `Equality` relation, they are created as instances of the composition of `EqAdaptor` and `Bit`. On the other hand, b3 is created as an instance of the composition of `TrAdaptor`, `EqAdaptor` and `Bit` (`TrAdaptor` is a mixin for `Trigger`), because it joins in both the `Equality` relation and the `Trigger` relation.

This solution is modular because the implementation of relations may be adapted to other implementations of Bit objects, if they implement the methods declared in `eqI`. Of course, the implementation of the Bit objects may be reused in other contexts. Furthermore, this solution is scalable because we may add new Bit objects easily via `join` methods declared in the relations. Adding new relations is also easy.

One of the keys of this solution is using mixin `EqAdaptor` that abstracts the operations in which `Equality` is interested, and ability to use the name `EqAdaptor` as a type name in formal parameters and field declarations. For example, we may also write the same example by using generics [2]; using generics, we may declare a mixin as a generic class whose superclass is a type parameter. However, a generic class is not a type; therefore, we must declare the `join1` method in `Equality` as

```
<T extends Bit> public void join1(EqAdaptor<T> e) { .. }
```

which makes the program more verbose. Furthermore, sometimes we should explicitly pass an actual type to the type parameter `T` that cannot be inferred in a method invocation; e.g., we will write the method invocation as `e2.<Bit>join2(b3)`. By allowing to use the name of mixin as a type, we may avoid this prolixity.

Another key of this solution is using `this` inside mixins. We use this feature in the method invocation `e.getOpponent(this)` in `EqAdaptor`. At first, this feature looks trivial; however, it is not. Using `this` inside mixins triggers some troublesome problems. For example, Jam [4], one of the extensions of Java with mixins, does not allow it. McJava is designed to safely use `this` inside mixins. This safety is discussed in Chapter 3. We will return to the relationship between our approach and other mixin-related systems in Chapter 7.

2.7 Summary

In this chapter, we have overviewed the design of McJava. McJava supports a mixin to be explicitly declared with a name, introducing additional modularity to Java. In McJava, a mixin name is also used as a type. The McJava's advanced features of higher order mixins and mixin-based subtyping promote reusability of programs still further. Besides simple examples such as `Color` and `Font`, this chapter also illustrates a more interesting example of integrated systems.

Chapter 3

Core McJava: A Core Calculus of McJava

Type soundness is one of the most basic properties of programming languages. In Chapter 2, we have designed McJava, an extension of Java with mixins. To investigate whether the property of type soundness still holds in McJava, we should carefully study how the features described in Chapter 2 interact with the existing constructs of Java.

This chapter presents 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 [34], a minimum core language of Java. FJ is a very small subset of Java, focusing on just a few key constructs that characterize the Java type system. FJ restricts on Java syntax so that FJ constructors always take the same stylized form; i.e., 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 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 not supported by FJ either.

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

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

Figure 3.1: Abstract syntax of Core McJava

McJava and its type soundness theorem.

3.1 Syntax

The abstract syntax of Core McJava is given in Figure 3.1. In this chapter, the metavariables d and e range over expressions; K_C and K_X range over constructor declarations; m and n range 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.

Core McJava imposes some syntactic restrictions for simplicity. First, a mixin in Core McJava must have exactly one constructor declaration, because it is the only place where assignments may appear. A constructor in mixin may be considered as an init function explained in section 2.5 that is implicitly invoked when a composition of the mixin is instantiated. Second, 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, that is, which method to be invoked is determined 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, namely the overloading problem in Jam (see Chapter 7). Because of these conditions, 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 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 $\#(\bar{x})$. An empty sequence is denoted by $\cdot$. Similarly, we write "$\bar{T}\ \bar{f}$" as a shorthand for "$T_1\ f_1, \cdots, T_n\ f_n$", "$\bar{T}\ \bar{f}\texttt{;}$" as a shorthand for "$T_1\ f_1\texttt{;}\ \cdots T_n\ f_n\texttt{;}$", "$\texttt{this}.\bar{f} = \bar{f}\texttt{;}$" as a shorthand for "$\texttt{this}.f_1 = f_1\texttt{;} \cdots \texttt{this}.f_n = f_n\texttt{;}$", and $\bar{X}$ as a shorthand for $X_1 :: \cdots :: X_n$.

As in Figure 3.1, 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 an 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 3.2, 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.

3.2 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

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

$$\bar{S} :: T :: \bar{U} \;<:\; \bar{S} :: \bar{U} \qquad \text{(S-COMP)}$$

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

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

Figure 3.2: Subtype relation

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 inference 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.

3.3 Auxiliary functions

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 3.3, 3.4 and 3.5.

The fields of type T, given in Figure 3.3, written *fields*(T), is a sequence $\bar{T}\ \bar{f}$ pairing the type of each field with its name. If T is a class, *fields*(T) is a sequence for all the fields declared in class T and all of its superclasses. If T is a mixin, *fields*(T) is a sequence for all the fields declared in that mixin. If T is a composition, *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 of *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.

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

$$\frac{\texttt{class}\ C\ \texttt{extends}\ \bar{X} :: D\ \{\bar{T}\ \bar{f}\texttt{;}\ K_C\ \bar{M}\} \qquad \mathit{fields}(\bar{X} :: D) = \bar{S}\ \bar{g}}{\mathit{fields}(C) = \bar{S}\ \bar{g},\ \bar{T}\ \bar{f}}$$

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

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

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

Figure 3.3: Field lookup

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 3.4 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 (Figure 3.5). As mentioned earlier, in contrast with FJ, method overloading is allowed in Core McJava.

3.4 Typing

The typing rule for compositions is given in Figure 3.6. 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

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

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

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

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

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

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

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

Figure 3.4: Method type lookup

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

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

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

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

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

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

Figure 3.5: Method lookup

$$\dfrac{\begin{array}{c} \mathit{fields}(X) \cap \mathit{fields}(T) = \emptyset \qquad \texttt{interface}\ I\ \{\bar{M}_I\} \\ \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ \ldots\ \ \bar{M}\ \} \\ \forall S\ m(\bar{T}\ \bar{x})\{\ldots\} \in \bar{M} \quad \mathit{mtype}(m, \bar{T}, X) = \mathit{mtype}(m, \bar{T}, T) \text{ or} \\ \mathit{mtype}(m, \bar{T}, T) = \texttt{nil} \\ \text{If } T \text{ is a composition } \bar{X} :: C, \text{ then} \\ \forall (U\ n(\bar{S}\ \bar{x})) \in \bar{M}_I\ \mathit{mtype}(n, \bar{S}, I) = \mathit{mtype}(n, \bar{S}, T) \end{array}}{X :: T \text{ ok}} \quad \text{(T-COMP)}$$

Figure 3.6: Well-formed composition

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 is straightforward and omited in this Figure.

Figure 3.7 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 expression. 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 3.8 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 overridden 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.

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

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

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

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

$$\frac{\Gamma \vdash e_0 : S \qquad S <: T \qquad T \text{ ok}}{\Gamma \vdash (T)e_0 : T} \qquad \text{(T-UCAST)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad T <: S \qquad T \neq S \qquad T \text{ ok}}{\Gamma \vdash (T)e_0 : T} \qquad \text{(T-DCAST)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : S \qquad T \not<: S \qquad S \not<: T \qquad T \text{ ok} \\ \mathit{stupid\ warning}\end{array}}{\Gamma \vdash (T)e_0 : T} \qquad \text{(T-SCAST)}$$

Figure 3.7: Expression typing

3.5 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 3.9. There are three reduction rules, one for field access, one for method invocation, and one for casting. 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

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

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

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

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

Figure 3.8: Well-formed definitions

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

3.6 Properties

We show that Core McJava is type sound. Intuitively, the step of proving Core McJava type soundness theorem is almost the same as that of FJ, but details

Computation:

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

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

$$\frac{\bar{X} :: C \ \ <: \ T}{(T)\texttt{new}\ \bar{X} :: C(\bar{e}) \longrightarrow \texttt{new}\ \bar{X} :: C(\bar{e})} \qquad \text{(R-CAST)}$$

Congruence:

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

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

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

$$\frac{e_i \longrightarrow e_i^{'}}{\texttt{new}\ \bar{X} :: C(\cdots, e_i, \cdots) \longrightarrow \texttt{new}\ \bar{X} :: C(\cdots, e_i^{'}, \cdots)} \qquad \text{(RC-NEW)}$$

$$\frac{e_0 \longrightarrow e_0^{'}}{(T)e_0 \longrightarrow (T)e_0^{'}} \qquad \text{(RC-CAST)}$$

Figure 3.9: Operational semantics

vary a little. We start by some lemmas used in the proof of type soundness.

Lemma 3.6.1 *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 3.6.2 *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. Note that whether m with argument types $\bar{T}$ is defined in C or not, *mtype*$(m, \bar{T}, C) =$ *mtype*$(m, \bar{T}, \bar{X} :: D)$ where `class` C `extends` $\bar{X} :: D \{\ldots\}$. Similarly, note that whether m with argument types $\bar{T}$ is defined in X or not, *mtype*$(m, \bar{T}, X :: T) =$ *mtype*$(m, \bar{T}, X)$ (see the rule T-COMP). □

Lemma 3.6.3 *If* $\Gamma, \bar{x} : \bar{S} \vdash e : U$*,* $\Gamma \vdash \bar{d} : \bar{R}$ *where* $\bar{R} <: \bar{S}$*, then* $\Gamma \vdash [\bar{d}/\bar{x}]e : T$ *for some* $T <: U$*.*

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

Case T-VAR.

$$e = x \quad U = \Gamma(x)$$

If $x \notin \bar{x}$, then the conclusion is immediate, since $[\bar{d}/\bar{x}]x = x$. If $x = x_i$, and $U = S_i$, then letting $R_i = T$ finishes the case because $[\bar{d}/\bar{x}]x = [\bar{d}/\bar{x}]x_i = d_i$, $d_i : R_i$ and $R_i <: S_i = U$.

Case T-FIELD.

$$e = e_0.f_i \qquad \Gamma, \bar{x} : \bar{S} \vdash e_0 : \bar{X} :: C$$
$$\textit{fields}(\bar{X} :: C) = \bar{T}\ \bar{f} \quad U = T_i$$

By the induction hypothesis, there is some T_0 such that $\Gamma \vdash [\bar{d}/\bar{x}]e_0 : T_0$ and $T_0 <: \bar{X} :: C$. Then, by Lemma 3.1, *ftype*$(f_i, T_0) =$ *ftype*$(f_i, \bar{X} :: C)$. Therefore, by the rule T-FIELD, $\Gamma \vdash ([\bar{d}/\bar{x}]e_0).f_i : T_i$.

Case T-INVK.

$$e = e_0.m(\bar{e}) \qquad \Gamma, \bar{x} : \bar{S} \vdash e_0 : \bar{X} :: C \quad \textit{mtype}(m, \bar{V}, \bar{X} :: C) = U$$
$$\Gamma, \bar{x} : \bar{S} \vdash \bar{e} : \bar{U} \quad \bar{U} <: \bar{V}$$

By the induction hypothesis, there are some T_0 and $\bar{X}$ such that

$$\begin{array}{ll} \Gamma \vdash [\bar{d}/\bar{x}]e_0 : T_0 & T_0 \ <: \ \bar{X} :: C \\ \Gamma \vdash [\bar{d}/\bar{x}]\bar{e} : \bar{T} & \bar{T} \ <: \ \bar{U} \end{array}$$

By Lemma 3.2, $\mathit{mtype}(m, \bar{V}, T_0) = \mathit{mtype}(m, \bar{V}, \bar{X} :: C) = U$. Then, by S-TRANS, $\bar{T} \ <: \ \bar{V}$. Therefore, by the rule T-INVK, $\Gamma \vdash [\bar{d}/\bar{x}]e_0.m([\bar{d}/\bar{x}]\bar{e}) : U$.

Case T-NEW.

$$\begin{array}{ll} e = \texttt{new}\ \bar{X} :: C(\bar{e}) & \mathit{fields}(\bar{X} :: C) = \bar{U}\ \bar{f} \\ \Gamma, \bar{x} : \bar{S} \vdash \bar{e} : \bar{T} & \bar{T} \ <: \ \bar{U} \end{array}$$

By the induction hypothesis, there are some $\bar{V}$ such that $\Gamma \vdash [\bar{d}/\bar{x}]\bar{e} : \bar{V}$ and $\bar{V} \ <: \ \bar{T}$. Then, by the rule S-TRANS, $\bar{V} \ <: \ \bar{U}$. Therefore, by the rule T-NEW, $\Gamma \vdash \texttt{new}\ \bar{X} :: C([\bar{d}/\bar{x}]\bar{e}) : \bar{X} :: C$.

Case T-UCAST.

$$e = (U)e_0 \quad \Gamma, \bar{x} : \bar{S} \vdash e_0 : T \quad T \ <: \ U$$

By the induction hypothesis, there are some V such that $\Gamma \vdash [\bar{d}/\bar{x}]e_0 : V$ and $V \ <: \ T$. Then, by the rule S-TRANS, $V \ <: \ U$. Therefore, by the rule T-UCAST, $\Gamma \vdash (U)([\bar{d}/\bar{x}]e_0) : U$.

Case T-DCAST.

$$e = (U)e_0 \quad \Gamma, \bar{x} : \bar{S} \vdash e_0 : T \quad U \ <: \ T \quad U \neq T$$

By the induction hypothesis, there are some V such that $\Gamma \vdash [\bar{d}/\bar{x}]e_0 : V$ and $V \ <: \ T$. If $V \ <: \ U$ or $U \ <: \ V$, then $\Gamma \vdash (U)([\bar{d}/\bar{x}]e_0) : U$ by the rule T-UCAST or T-DCAST, respectively. On the other hand, by the rule T-SCAST, $\Gamma \vdash (U)([\bar{d}/\bar{x}]e_0) : U$ (with a *stupid warning*).

Case T-SCAST.

$$e = (U)e_0 \quad \Gamma, \bar{x} : \bar{S} \vdash e_0 : T \quad U \ \not<: \ T \quad T \ \not<: U$$

By the induction hypothesis, there are some V such that $\Gamma \vdash [\bar{d}/\bar{x}]e_0 : V$ and $V \ <: \ T$. If $V \ \not<: \ U$, then, by the rule T-SCAST, $\Gamma \vdash (U)([\bar{d}/\bar{x}]e_0) : U$ (with a *stupid warning*). If $V \ <: \ U$, then, by the rule T-UCAST, $\Gamma \vdash (U)([\bar{d}/\bar{x}]e_0) : U$.
□

Lemma 3.6.4 *If* $\Gamma \vdash e : T$ *where* Γ *does not include* x*, then* $\Gamma, x : U \vdash e : T$.

Proof. Straightforward induction. □

Lemma 3.6.5 *If* $mtype(m, \bar{U}, \bar{X} :: C) = U$ *and* $mbody(m, \bar{U}, \bar{X} :: C) = \vec{x}.e$*, then, for some* U_0 *with* $\bar{X} :: C <: U_0$*, 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*. In the base case (where m is defined in $CT(T_0)$), it is easy to prove by the rule T-CMETHOD, if T_0 is a class type, or by the rule T-XMETHOD, if T_0 is a mixin type. The induction step is also straightforward. □

From the lemmas established above, we derive the type soundness theorem for Core McJava:

Theorem 3.6.1 (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 $e \longrightarrow e'$.

Case R-FIELD.

$$e = (\texttt{new}\ \bar{X}(C)(\bar{e})).f_i \quad e' = e_i \quad fields(\bar{X}(C)) = \bar{U}\ \bar{f}$$

By the rule T-FIELD, we have $\Gamma \vdash \texttt{new}\ \bar{X} :: C(\bar{e}) : \bar{Y} :: D$, $T = U_i$ for some $\bar{Z} :: E$. Then, by the rule T-NEW, we have $\Gamma \vdash \bar{e} : \bar{T}$, $\bar{T} <: \bar{U}$, $\bar{Y} :: D = \bar{X} :: C$. In particular, $\Gamma \vdash e_i : T_i$, finishing the case, since $T_i <: U_i$.

Case R-INVK.

$$\begin{array}{ll} e = (\texttt{new}\ \bar{X} :: C(\bar{e}).m(\bar{d}) & mbody(m, \bar{T}, \bar{X} :: C) = \bar{x}.e_0 \\ e' = [\bar{d}/\bar{x}, (\texttt{new}\ \bar{X} :: C(\bar{e}))/\texttt{this}]e_0 & \end{array}$$

By the rule T-INVK and T-NEW, we have

$$\begin{array}{ll} \Gamma \vdash \texttt{new}\ \bar{X} :: C : \bar{X} :: C & mtype(m, \bar{T}, \bar{X} :: C) = T \\ \Gamma \vdash \bar{d} : \bar{U} & \bar{U} <: \bar{T} \end{array}$$

for some $\bar{U}$ and $\bar{T}$. By Lemma 3.5, $\bar{x} : \bar{T}, \texttt{this} : T_0 \vdash e_0 : S$ for some T_0 and S where $\bar{X} :: C <: U_0$ and $S <: T$. By Lemma 3.4, $\Gamma, \bar{x} : \bar{T}, \texttt{this} : T_0 \vdash e : S$. Then, by Lemma 3.3, $\Gamma \vdash [\bar{d}/\bar{x}, (\texttt{new}\ \bar{X} :: C(\bar{e}))/\texttt{this}]e_0 : V$ for some $V <: S$.

Then we have $V <: T$ by transitivity of $<:$. Finally, letting $V = T'$ finishes this case.

Case R-CAST.

$$e = (U)(\texttt{new}\ \bar{X} :: C(\bar{e})) \quad \bar{X} :: C(\bar{e}) <: U \quad e' = \texttt{new}\ \bar{X} :: C(\bar{e})$$

Because of the assumption $\bar{X} :: C <: T$, the proof of $\Gamma \vdash (T)\texttt{new}\ \bar{X} :: C(\bar{e}) : U$ must end with the rule T-UCAST. By the rules T-UCAST and T-NEW, we have $\Gamma \vdash (U)\texttt{new}\ \bar{X} :: C(\bar{e}) : U$.

The cases for congruence rules are easy. □

Theorem 3.6.2 (Progress) *Suppose e is a well-typed expression.*

1. *If e includes* $\texttt{new}\ \bar{X} :: C(\bar{e}).f$ *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* $\texttt{new}\ \bar{X} :: C(\bar{e}).m\texttt{<}\bar{T}\texttt{>}(\bar{d})$ *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.

Proof. If e has $\texttt{new}\ \bar{X} :: C(\bar{e}).f$ as a subexpression, by well-typedness of the subexpression, it is easy to check that *fields*$(\bar{X} :: C)$ is well defined and f appears in it. Similarly, if e has $\texttt{new}\ \bar{X} :: C(\bar{e}).m\texttt{<}\bar{T}\texttt{>}(\bar{d})$ as a subexpression, it is also easy to show *mbody*$(m, \bar{T}, \bar{X} :: C) = \bar{x}.e_0$ and $\#(\bar{x}) = \#(\bar{d})$, since *mtype*$(m, \bar{T}, \bar{X} :: C) = U$ where $\#(\bar{x}) = \#(T)$. □

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

Theorem 3.6.3 (Core McJava Type Soundness) *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow^* e'$ *with* e' *a normal form, then* e' *is either (1) a value* v *of* e *with* $\emptyset \vdash v : U$ *and* $U <: T$, *or (2) an expression containing* $(U)\texttt{new}\ T(\bar{e})$ *where* $U \not<: T$.

Proof. Immediate from Theorem 3.1 and 3.2. □

3.7 Summary

In this chapter, we have defined Core McJava, a core calculus of McJava. The definition contains all the key constructs that characterize the McJava type system such as mixin declarations, mixin composition operator ::, mixin-based subtyping, and so on (it also contains method overloading). The dynamic semantics is defined with simple reduction semantics. After the definition, we have proven type soundness of Core McJava that ensures McJava type system is sound.

Chapter 4

Implementation of McJava

We have designed an extension of Java with mixins, and formalize it at an abstract level. To design a useful language, however, we should also think about compilation of the language into the format that the run-time system can efficiently execute. Another criteria for usefulness is compatibility. The new language should run efficiently on the existing standard platforms, and existing libraries should also be able to be used without any changes.

In this chapter, we discuss a compilation strategy of McJava. Our McJava compiler compiles McJava programs into correct Java programs (i.e. they are guaranteed to be compiled into Java byte code by using Java compilers) thus making it runnable on the standard Java virtual machine.

4.1 Outline of the Compilation Strategy

Besides the class system provided by Java, McJava provides the construct of mixins and the mechanism of mixin-based composition. The problem is how to map these constructs into Java. It may appear that the body of a mixin can be easily translated into a Java class; however, handling of composition is not so simple. As mentioned in the previous chapters, the subtype relation defined among compositions is very flexible. Therefore, how to map the compositions into a single inheritance language (i.e. Java) is a non-trivial problem.

Figure 4.1 outlines how the composition `M::C`, where `M` is a mixin and `C` is a class, is translated into Java hierarchies; it generates an interface hierarchy and a class hierarchy (we use a symbol ◁—— to denote subtype relation). Each

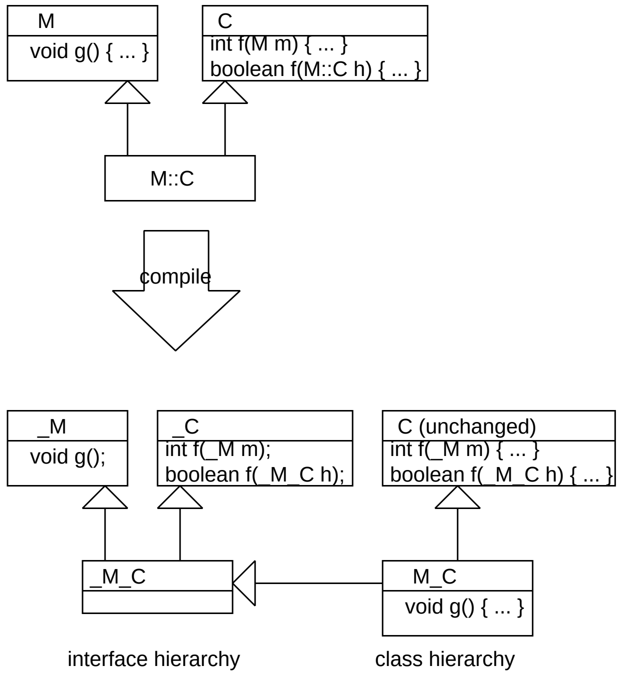

Figure 4.1: Translation into Java classes

interface in the interface hierarchy corresponds to each type in McJava. In this hierarchy, McJava subtyping is preserved (note that in McJava an interface may inherit from multiple interfaces). The top level interfaces (i.e. `_M` and `_C`) declare methods extracted from definitions contained in McJava classes and mixins. At the same time, the definitions contained in mixins are copied into the class hierarchy. This class hierarchy preserves the inheritance relationship of mixin-based composition; that is, the relation of "`M` inherits from `C`" is translated into "`M_C` inherits from `C`." For each class name, we use a string concatenating the name of mixin and the name of superclass with a special character '_'.[1]

In order to gain a more concrete image of the translation process, we use an example code shown in Figure 4.2. We also note that this program is ill-typed in Jam [4], which is discussed in Chapter 7.

Before executing the translation, the McJava compiler prepares a table that consists of names of classes and mixins, and their declarations, shown in Table

[1]This implies that our compilation triggers code duplication.

```
class C {
  int f(M m) { ... }
  boolean f(M::C h) { ... }
}
interface I { /* empty */ }
mixin M requires I {
  void g() {
    int i = new C().f(this);
    ...
  }
}
class Test {
  public static void main(String args[]) {
    new M::C().g();
  }
}
```

Figure 4.2: An example program illustrating the compilation

4.1. The translator processes all the entries of this table to generate Java classes and interfaces.

Table 4.1: Prepared class table

Name	Declaration body
`C`	`class C { int f(M m) { ... }` `boolean f(M::C h) ...`
`M`	`mixin M requires I { void g() { ...`
`Test`	`class Test { public static void main ...`

At the first step, the translator creates a file `C.java` from the entry `C`. Then, it writes the body of class declaration into that file. In the beginning, the translator just copies the body of class `C` into `C.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 a new interface `_M_C`, as shown in Figure 4.1, and replaces the occurrence of `M::C` with the *interface type* `_M_C` (actually, the occurrence of mixin type `M` is also replaced with the interface type `_M`):

```
class C {
  int f(_M m) { ... }
  boolean f(_M_C h) { ... }
}
```

The class `M_C` extends the class `C` and implements the interface `_M_C` that extends interfaces `_M` and `_C`. Those interfaces contain interface method declarations extracted from the mixin `M` and the class `C`, respectively. The class `M_C` contains definitions copied from the mixin `M`:

```
interface _M { void g(); }
interface _C { int f(_M m); boolean f(_M_C h); }
interface _M_C extends _M, _C { }
class M_C extends C implements _M_C {
  void g() {
    int i = new C().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 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[2].

At the second step, the translator processes the entry `M`. Because mixin implementation is never executed unless its composition is instantiated, the translator only extracts the interface from the mixin `M` to generate interface `_M`.

[2]We comment that this kind of fix may work as well for Jam to relax a little bit the copy principle.

That interface has been generated in the previous step; therefore, in this case this step is actually skipped.

Finally, the translator processes the entry `Test`. In the body of class `Test`, the translator encounters the composition `M::C` again. However, in this case it is used as an instance creator. Since the interface type cannot be used for this purpose, in this case we replace `M::C` with the *class* `M_C`:

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

Note that the translation does not change any occurrence of class types and interface types; e.g. the occurrence of `String` is left unchanged. This property is to guarantee backward compatibility to the existing libraries. We would not like to make any changes on the libraries that contain only pure Java constructs.

So far, a simple case is explained. We now describe a more general case:

- A composition $X_1::\cdots::X_n::C$, where each X_i $(i \in 1 \cdots n)$ is a mixin and C is a class, is translated into a class $X_1\_\cdots\_X_n\_C$ that implements the interface $\_X_1\_\cdots\_X_n\_C$ and extends the class $X_2\_\cdots\_X_n\_C$. The body of the class $X_1\_\cdots\_X_n\_C$ is a copy of X_1. We say that the interface $\_X_1\_\cdots\_X_n\_C$ corresponds to the composition $X_1::\cdots::X_n::C$. Similally, we say that the class $X_1\_\cdots\_X_n\_C$ corresponds to the composition $X_1::\cdots::X_n::C$.

- The interface $\_X_1\_\cdots\_X_n\_C$ extends all the interfaces that correspond to each of $X_1::\cdots::X_n::C$'s immediate super types.

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

Figure 4.3 outlines how the compositions `N::M::C` and `N::C`, where `N` is a mixin, are translated into Java hierarchies. In this case, the body of mixin `N`

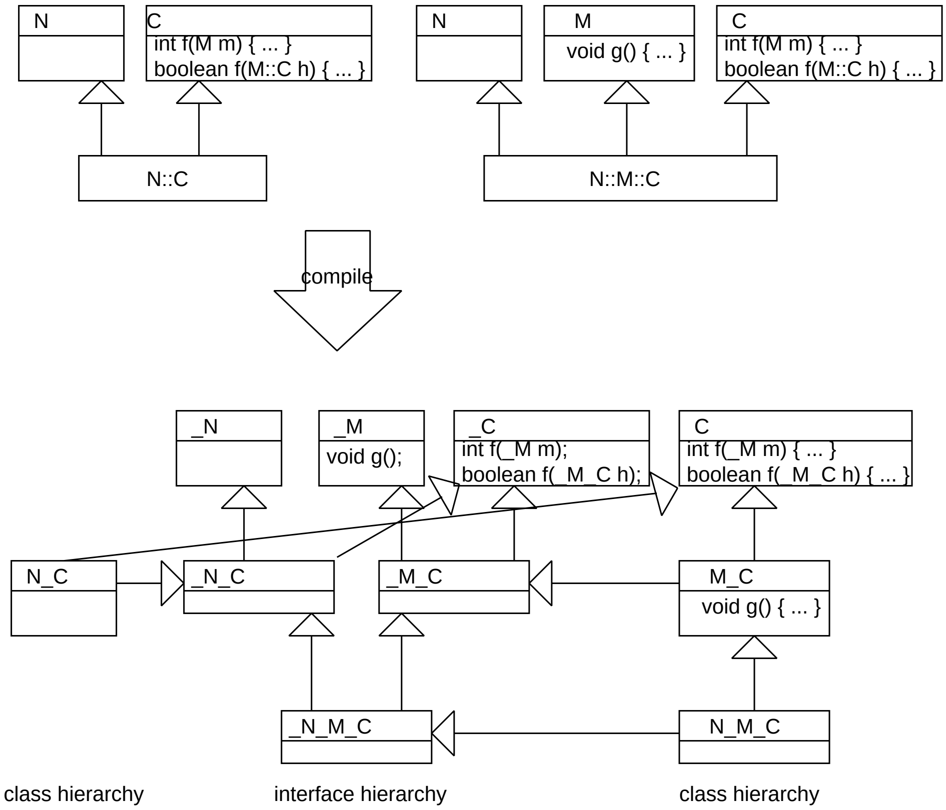

Figure 4.3: Translation into Java classes (a complex case)

is copied into the body of `N_M_C` and `N_C`. These classes implement the corresponding interface, respectively.

4.2 Evaluating the compilation

We sketch that this translation preserves behavior of the McJava program. First of all, all the composition types are replaced with corresponding interface types, and subtype relations are preserved among them. Each class in the class hierarchy also corresponds to the McJava composition type, and subtype ralations are also preserved, because each class implements the corresponding interface. The class hierarchy is used only for the instance creation. The order of method invocation on these instances is also preserved, because the translated class hierarchy preserves the inheritance relation among constituents of the composition.

One may wonder why there is no subtype relation between _C and C. To make the compiler backward compatible to the existing Java libraries, we should not make any change on the structure of class hierarchies. Therefore, we should not add another `implements` relation between _C and C.

But a question still remains. Suppose we have the following McJava code fragment:

```
M::C m = new M::C();
C c = m;
```

This code is translated to the following Java code

```
_M_C m = new M_C();
C c = m;
```

that results in a compile error because C and _M_C are not comparable. However, we can also avoid this error by injecting the type-cast as follows:

```
_M_C m = new M_C();
C c = (C)m;
```

So far, our McJava compiler is backward compatible to standard Java compilers[3]. That is, every Java program that can be compiled with a standard Java compiler may also be compiled with the McJava compiler. Actually, following the above algorithm, the McJava compiler does *nothing* when it consumes a standard class written in pure Java. This means that our compiler does not degrade run-time performance of Java. Furthermore, in our approach, a mixin composition is translated into a class hierarchy whose depth is exactly the same as the depth of the correspinding mixin composition. This implies that the run-time performance of mixin-based composition is also reasonable; e.g., the cost of a method dispatch on an instance of a composition is the same as the cost of method dispatch on the corresponding class inheritance chain.

At the moment, we have developed a preliminary version of McJava compiler that has some restrictions including that it still does not have the capability of accessing Java standard libraries. However, since our compilation

[3]Except for that McJava reserves keywords `mixin` and `requires`

scheme leaves all the program pieces that do not contain any McJava specific statements unchanged, we may easily add an ability to use the existing Java libraries to McJava compiler. The current prototype version of McJava compiler is written in the Objective Caml language. We are now planning to develop more practical compiler by using a convenient tool such as Polyglot [45], a framework for developing a compiler of an extension of Java.

4.3 A Sketch for Separate Compilation

Current McJava compilation procedure does not support separate compilation. This does not necessarily mean that it is impractical. Actually there are some practical systems that do not support separate compilation such as templates on some C++ compilers and AspectJ compiler [38].

It is clear, however, that support for separate compilation is very helpful to distribute binary form of mixins. Fortunately, McJava type system allows separate type checking of each mixin; therefore, we may pursue a way for separate compilation. For this purpose, we think that introducing a *linker* that composes the binary mixins before load time will solve the problem[4]. Instead of analyzing the whole program, this compiler will compile a mixin to a class whose superclass has a dummy implementation of the required interface. The linker links classes and mixins to create binary form of compositions by manipulating class files generated by the compiler.

4.4 Summary

In this chapter, we have discussed how McJava programs are efficiently translated into Java programs. This gives an assurance that McJava programs are efficiently runnable on the standard Java virtual machine. Moreover, we may lead a way for developing a more practical compiler that enable us to use the existing Java libraries in McJava programs, because our compilation strategy does not change any pure Java code. Owing to its ability to type-check mixins separately, we may also find a way to separate compilation.

[4]The idea is taken from Jiazzi [42].

Chapter 5

An Advanced Mechanism of Method Dispatch

Sometimes, a new programming language construct, which solves some problems, also produces new problems. One problem that mixin-based composition raises is known as *accidental overriding* [2]. Unlike inheritances in many object-oriented languages where a subclass explicitly declares its superclass, in mixin-based composition, a mixin does not know which superclass the mixin will be composed with. Therefore, when a user of a mixin (who will be different from the implementor of that mixin) tries to compose it with some other classes, it is possible that a method declared in the mixin accidentally overrides a method declared in the superclass.

This chapter presents a new mechanism of method dispatch that solves the accidental overriding problem and how to implement it in McJava compiler.

5.1 The Problem of Accidental Overriding

In general, there are two kinds of overriding: *intentional* overriding and *accidental* overriding. In the case of intentional overriding, we know that a superclass has a method that will be overridden. In this case, we explicitly declare methods imported from the superclass (e.g. as explained in the previous chapters, we can use `requires` clause for this purpose in McJava), then override them in a mixin. In the case of accidental overriding, on the other hand, we do not know that the superclass has a method whose name and formal parameter

```
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 5.1: Accidental Overriding in McJava

types are the same as those of a method declared in the mixin. This overriding is harmful because it accidentally changes the behavior [40] of the superclass.

In Figure 5.1, we illustrate the problem of accidental overriding by using McJava programming language. This figure declares a class `Person` that represents core attributes of a person (in this example, it only contains an attribute corresponding to the name of person). The figure also declares two mixins, `Employee` and `Student`. The class `Person` can be composed with mixin `Employee`, because it implements the interface that the mixin `Employee` requires (i.e. `String name()` method). The imported methods declared in the `requires` clause is referred in the body of mixin; i.e., `super.name()` is called inside `Employee.name()`. In other words, `Employee` intentionally overrides the method `String name()`. In Figure 5.1, this composition, `Employee::Person` 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.

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

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* [29, 2, 42]. 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

[1]The source of the term "method combination" is CLOS [37].

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* [30].

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

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

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

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

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

McJava allows a composition `Id::Student::Employee` to be a subtype of `Id::Employee`, which means, in McJava, subtype relations are not restricted to the immediate inheritance relations. In the above case, local variable `e` has type `Id::Employee`; therefore, the executed code of `super.getID()` in `Id.getID()` should be `Employee.getID()`.

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

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

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

5.2 Selective Method Combination

To tackle the problem, we propose a new approach to method lookup that solves the accidental overriding problem. Our approach allows *selective method combination*; that is, if we have multiple candidates for method call to `super`, we can select which method to be called. This selection is also achieved by using static type information of the receiver. Our approach is actually an extension of hygienic mixins. However, as we have seen, since McJava provides flexible mixin-based subtyping, adopting hygienic mixins to McJava is actually a non-trivial issue. In McJava, an immediate superclass of a mixin in the runtime inheritance chain may be different from the statically known superclass thus requiring more sophisticated treatment in invoking a superclass's method.

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 5.2):

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

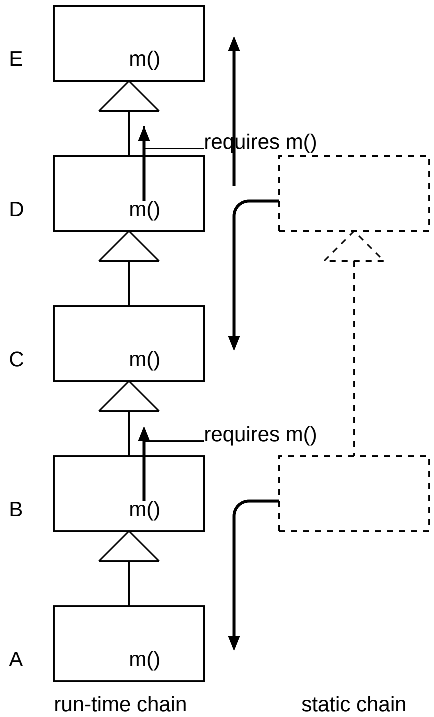

Figure 5.2: New method lookup in McJava

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 5.2. 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 5.2), 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 5.2, 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 5.2, `A` does not require a method `void m()`; therefore, the resolved method is `B.m()`.

4. The method lookup then searches the superclass's method called on `super`. This search goes *up* on the run-time inheritance chain until it reaches the starting point (`B` in Figure 5.2). 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 5.2, `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 5.2, 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 5.2 are `B.m()`, `D.m()` and `E.m()`. In other words, the method combination from `A.m()`, `B.m()`, `C.m()`, `D.m()` and `E.m()` with a static scope `B::D` is [`B.m()`, `D.m()`, `E.m()`].

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

5.3 Implementation Issues

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

McJava compilation strategy is explained in Chapter 4. In this chapter, we briefly sketch how the renaming of methods works in the compilation. Figure 5.3 and 5.4 shows the translated Java code from the definitions in Figure 5.1 and `Id` in section 5.1:

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

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

The method name invoked externally is also changed. For example, the declaration of `processIdOfEmployee` in section 5.1 becomes the following dec-

```
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; }
}
```

Figure 5.3: Compiled code of Figure 5.1 and `Id` (1)

```
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 5.4: Compiled code of Figure 5.1 and `Id` (2)

laration:

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

5.4 Summary

In this chapter, we have shown how the mechanism of selective method combination addresses the problem of accidental overriding. Our approach guarantees that the most specific method from the view point of the statically known type of the receiver is guaranteed to be executed in the case that multiple methods coexist in the same object. This mechanism is implemented in the McJava compiler as source-to-source translation, by using the technique of method renaming.

Our approach may look specific only to be applied to McJava because it depends on McJava subtyping rules. However, some languages such as `gbeta` [26] allow similar mechanism as McJava.[2] We believe that the proposal of this paper can be applicable to such languages. Furthermore, as shown in the previous 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 chapter always arises and the proposed solution may be useful.

[2] We note the similarity and difference between McJava and `gbeta` in Chapter 7.

Chapter 6

Mixins and Other Language Features

So far, we have explored how to add mixin-based composition into the Java programming language. Besides our work, there are many researches on adding new constructs to the conventional Java. These researches are independent to mixins; how mixins interact with these constructs still remains as an open issue.

This chapter investigates how mixins are related with generics and `ThisType`. The mechanism of generics plays an important role on definition of polymorphic "collection classes" such as `Set` and `List`, which are monomorphic in the conventional Java. For example, while the monomorphic lists only guarantee that their elements are at most `Object`s (so when we use values stored in the list, we have to downcast them in order to do anything useful with them), we may create an instance of a list whose elements are guaranteed to be integer values:

```
List<Integer> li = new List<Integer>();
```

This feature is now included in Java; therefore, it is interesting to study relationship between generics and mixins.

There is also an interesting study on adding `ThisType` (or `ThisClass`) to Java [14]. `ThisType` is a name of type that refers to a type of `this`. This construct enhances extensibility of modules. For example, the conventional Java library includes an interface `Cloneable` that is implemented by classes

```
class List<T <: Object> {
  T head;
  List<T> tail;
  public List(T h, List<T> t) {
    head = h; tail = t;
  }
  ...
}
```

Figure 6.1: An example of generic class

which may be cloned. In the definition of the `clone()` method (declared in `Object`), however, we cannot predict which class implements this interface; therefore, a return type of `clone()` is declared as `Object`, that is a supertype of all the reference types. Since `Object` gives no useful information of the type of the object that `clone()` returns, programmers must downcast it to some expected type in order to do anything useful with it. `ThisType` construct compensates this limitation; we can declare `clone()` as follows:

```
ThisType clone();
```

Even though this feature is not included in the official version of Java, it provides much extensibility and it is interesting to study how this feature relates to mixins.

This chapter presents a design of a language that extends McJava with generics and `ThisType`. This chapter also demonstrates how expressive this language is by showing an example. With this language, we can actually construct a more flexible version of mixin layers [54].

6.1 Generics

Generics is a mechanism of applying parametric polymorphism in type systems of functional languages such as ML to object-oriented languages. It abstracts type information that appears in class declarations, method declarations, and interface declarations, by using *type parameters*. Figure 6.1 shows an example that declares a generic class `List`. In this figure, type parameters are declared between angle brackets (`<>`) following the class name `List`. We can use the class name `List` as a type with the form of `List<Integer>` or `List<String>` that means a list whose elements are guaranteed to be integer values or a list whose elements are guaranteed to be string values, respectively:

```
List<String> sl = new List<String>("foo",
                  new List<String>("bar", null));
String s = sl.head;
List<Integer> il = new List<Integer>(new Integer(10),
                    new List<Integer>(new Integer(20),
                                      null));
Integer i = il.head;
```

The right operand of `<:` in Figure 6.1 is an *upper bound* of the type parameter `T` (in this case, that is `Object`), which means a value of `T` is at most an `Object`. While C++ templates mechanism that does not support such upper bounds is sometimes called *unbounded polymorphism*, the generics in Java is called *bounded polymorphism*. Moreover, it actually allows type parameters to appear in upper bounds (*F-bounded polymorphism* [16]). It is known that this feature is useful for writing extensible programs (one example using it is found in [63]).

Like generic classes, we may also consider generic mixins, which are mixins whose type information in its declaration is abstracted by using type parameters. Furthermore, we may also use type parameters inside `requires` clause:

```
mixin Color<T <: Graphics> requires {
    void paint(T g);
  } {
  private int color;
  void paint(T g) {
    g.setColor(color);
    super.paint(g);
  }
  ...
}
```

```
class App<C <: Color, F <: Font> {
  List<C> colorList;
  List<F> fontList;
  List<C::F> colorFontList;
  ...
}
```

Figure 6.2: Composition of type parameters

As in the case of generic classes, we may also use the mixin name as a type with the form of `Color<Graphics>`. We may use it as if we have defined a mixin `Color` whose `paint` method's type parameter is `Graphics`; we can compose it with an other class that implements the method `void paint(Graphics g)`.

Conceptually, in class-based systems, type parameters may appear every place where types can be used[1]. Therefore, one may also wonder whether type parameters may be used as operands for mixin composition operator `::`. For example, Figure 6.2 declares a generic class `App` that contains an instance variable of list whose elements are guaranteed to be compositions of `C`, which is at most `Color`, and `F`, which is at most `Font`. By instantiating `App` as `App<Color::RGB, Font>`, we may get a list of elements featured by "color with RGB" and "font." This declaration of `App`, however, can be dangerous, because we may also instantiate `App` with `App<Color::Label, Font>`, which results in a list of elements of `Color::Label::Font`, but this composition is actually ill-formed.

[1]In GJ[11] (so as in Java5), using type parameters in constructor invocation and type casting is restricted.

One way to avoid this ill-formed composition is to forbid a composition containing type parameters. However, this approach is too restrictive, because it rejects all compositions containing type parameters, even when type parameters are assigned safe types. Actually, we may take more flexible approach. That is, we may allow the ill-formed composition, unless we create a *value* of it. For example, in the following code fragment, the compiler can report an error when it finds an instance creation of `Color::Label::Font`:

```
new App(Color::Label, Font).
  colorFontList.insert(new Color::Label::Font());
                                // compile error
```

In other words, an ill-formed composition can be considered as a *bottom type*, a subtype of all the types that has no inhabitants.[2]

To guarantee that there are no values of ill-formed composition, we still impose the following restrictions on the type system; constructor invocations must not contain type parameters, and type casts must not contain type parameters. These restrictions are not so arbitrary; they are actually imposed on Java5, although in Java5 these restrictions stem from the requirements for backward compatibility.

6.2 ThisType

`ThisType` [14] stands for the type of `this`. If a class `C` declares a method `m`, then when `m` is invoked on an object whose run-time type is `C`, any occurrences of `ThisType` in the `m`'s type signature may be safely assumed to be `C`. When `m` is inherited in a subclass of `C`, or `C` is composed with some mixins, then the occurrences of `ThisType` in `m` are assumed to have all the features of this subclass or composition, respectively. For example, suppose that `M` is a mixin, `mc` is an expression of type `M::C`, and `m` is a method declared in `C` with static type

```
m:  ThisType → void
```

Then, the type of `m` is considered to be

[2] Except for `null`, of course.

```
class C {
  void m(ThisType c) { ... }
}
mixin M requires { void m(ThisType c); } {
  void n() { ... };
  void m(ThisType c) { ...; c.n(); ...; }
}
class Main {
  static void boom(C c1, C c2) { c1.m(c2); }

  public static void main(String[] args) {
    boom(new M::C(), new C()); // error!
  }
}
```

Figure 6.3: An example of error using `ThisType`

`m:  M::C → void`

when the method invocation `mc.m(o)` is typechecked.

In order to ensure that the above method invocation is safe, we need to be able to determine the precise type that the receiver will have at run-time. Figure 6.3 shows why this property is needed. If the typecheckor does not report an error when it analyzes the line labeled "`error!`," then the evaluation of `c1.m(c2)` in the body of `boom` would send the message `n` to an object of type `C`, which has no such method.

To avoid this problem, we may give up the subtyping between `ThisType` of before the extension and that of after the extension. One way to do this is to introduce *exact types*, which guarantees that the type of run-time objects do not change while the computation [14]. Another instance of this problem and how to rule it out is discussed in section 6.4.

Note that, as in Figure 6.3, we may also use `ThisType` in the `requires` clause of mixin declarations, which means that the mixin `M` requires a method `m` whose formal parameter type is literally declared as `ThisType`.

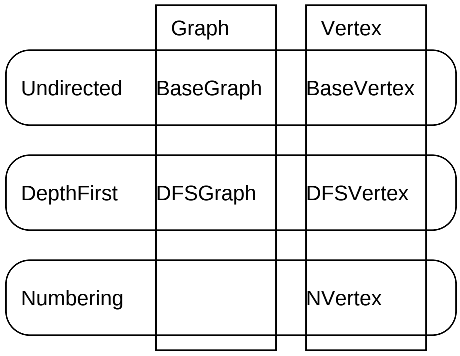

Figure 6.4: Layered Design

6.3 An Approach to Layered Design

In this section, we show an example program written in McJava extension with generics and `ThisType`. With the extension, we can write a flexible and extensible program for layered design [18]. Before we proceed, we introduce an existing programming method named *mixin layers*, because our approach presented in this section is an enhancement of this method.

6.3.1 Mixin Layers

Layered design that decomposes software as layers of functions is considered suitable for constructing a "family of programs" [48]. For example, Figure 6.4 shows an image of layered design for a graph traversal application.[3] In this example, there are two objects that participate in the application. These objects are instances of classes `Graph` and `Vertex` respectively; in Figure 6.4, each of these classes is depicted by a rectangle. There are also concerns that cross-cut these classes, such as a graph representation, a graph traversal algorithm, some kinds of processing on the graph (such as numbering of each vertex), and so on. These concerns are depicted as rounded rectangles in Figure 6.4.

By using layered design, we may easily extend the program with new features such as cycle checking, or we may easily replace a layer with another layer (e.g., we may replace `DepthFirst` with `BreadthFirst`). This is why the layered design can effectively construct family of programs.

[3]This figure has been used in [32], [66], and [54] to illustrate each work.

```
class UndirectedGraph {
  class Graph { ... };
  class Vertex { ... };
};
template <NextLayer>
class DepthFirst : public NextLayer {
  class Graph : public NextLayer::Graph { ... };
  class Vertex : public NextLayer::Vertex { ... };
};
template <NextLayer>
class Numbering : public NextLayer {
  class Vertex : public NextLayer::Vertex { ... };
  /* functions for numbering vertecies */
};
```

Figure 6.5: Mixin layers implementation using C++

Mixin layers [54] is one promising programming method for implementing layered design. Figure 6.5 shows an example of mixin layers that implements Figure 6.4; it uses C++ templates. A class `UndirectedGraph` implements a layer of "undirected graph" (`Undirected` in Figure 6.4); it declares `Graph` and `Vertex` as inner classes. A template `DepthFirst` implements a layer of graph traversal algorithm (in this case the depth first algorithm is used). The superclass of `DepthFirst` is a type parameter of templates, which means `DepthFirst` is a mixin.[4] `DepthFirst` also declares two inner classes `Graph` and `Vertex` whose superclasses are inner class members of `NextLayer` (a type parameter of `DepthFirst`); i.e., these inner classes are also mixins.

[4]The difference between mixins in McJava and mixins in C++ templates is discussed in Chapter 7.

One of the advantages of mixin layers is the modularity of each layer; e.g., we may compose `UndirectedGraph` with a traversal layer other than `DepthFirst` such as `BreadthFirst`, or we may compose `DepthFirst` with a graph layer other than `UndirectedGraph` such as `DirectedGraph`. Another advantage of mixin layers is its convenience for composing large scale layers; we may simply construct an application by composing each layer:

```
typedef Numbering<DepthFirst<UndirectedGraph>> App;
App::Graph *graph = new App::Graph();
```

Since we adopt a convention to use the same name for each inner class that appears in each layer, when the above composition is placed, these inner classes are also composed.

Cardone et al. argued that layered design and implementation is more suitable for program reuse and evolution than existing object-oriented frameworks [18]. Furthermore, by this modularity of layers and simplicity of composition, mixin layers are considered as an effective way for constructing software product lines [21].

6.3.2 Our Approach to Generic Mixin Layers

Despite its modularity, the method of mixin layers has some limitations. First, in mixin layers, it is rather difficult to implement *a series of layers*, because a mixin layer consists of only one module; i.e., in mixin layers, the elements in a layer (that are implemented as inner classes) cannot be modularized. It would be more convenient if we have modules that implement the elements of a layer, and we can construct a layer by using them. For example, we may have two kinds of implementation for graphs: adjacency-matrix representation that is suitable for dense graph, and adjacency-list representation that is suitable for sparse graph. Furthermore, we may also have multiple representations for vertex, such as colored vertex and uncolored vertex. It would be convenient if we can select a graph implementation from the variety of graph representations, and if we can also select a kind of vertex in the same way, for construction of a layer.

The second limitation of mixin layers is C++ specific; that is, it is impossible to typecheck each layer separately.

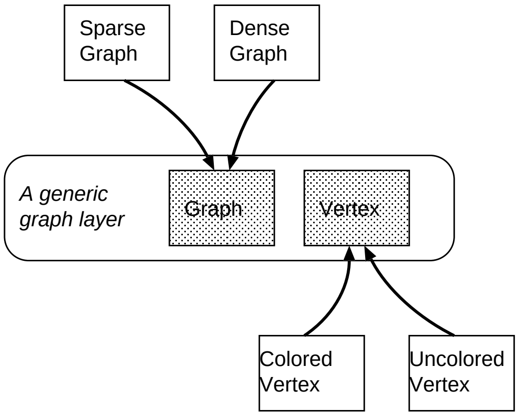

Figure 6.6: A generic graph layer

By using the constructs introduced in section 6.1 and 6.2, we may resolve the limitations imposed by mixin layers. Our approach actually provides additional modularity that allows implementation of *generic* mixin layers whose "inner classes" are parametrized.

Outline of our approach. Our approach makes it possible to implement a generic layer whose "inner classes" are parametrized, which means we may "inject" a separately developed inner classes to the parameters of the generic layer (Figure 6.6). This parameterization is achieved by using generic classes. A non-trivial issue on this separation is as follows; each inner class must still be able to refer to each other, even when they are separately developed. This means that the generic layer should have information about each inner class, and each inner class should also have information about the layer to access the information that the layer has. We may use F-bounded polymorphism for this purpose.

After constructing each layer by filling in the "holes" with the separately developed inner classes, we may simply combine them by mixin-based composition, as we saw in the mixin layers method. Furthermore, our approach allows separate typechecking of each layer. This checking is achieved by using `requires` clause of McJava. In checking, we should also take into account that some types in each layer will be extended after composition. To predict this extension, we may use the `ThisType` construct.

```
class GraphLayer<&G <: Graph<GraphLayer<G,V>>,
                    &V <: Vertex<GraphLayer<G,V>>> {
    ...
}
```

Figure 6.7: A generic graph layer

In the rest of this section, we show our approach in detail.

Parameterizing "inner classes." Figure 6.7 shows a generic graph layer. The generic class `GraphLayer` declares type parameters `G` and `V` whose upper bounds are declared as generic interfaces `Graph` and `Vertex`, respectively.

These type parameters provide places that the separately developed "inner classes" will be stored; however, they are not like inner classes in that it is impossible to access these type parameters from the outside of `GraphLayer`. For this purpose, we introduce a new convenient mechanism of allowing this access. If a type parameter is annotated with `&`, this type parameter can be referred as a field of the class:

```
GraphLayer<MyGraph,MyVertex>.G graph = ...;
```

This mechanism is like virtual types [35, 61, 51], although virtual types provide mechanism of declaring types as *instance variables*.

Note that we the put type parameters `G` and `V` as arguments for generic interfaces `Graph` and `Vertex`, which restricts on the very type parameters that are passed to them (F-bounded polymorphism).

Implementing the "inner classes." Figure 6.8 shows an implementation of a graph that will be bound to the type parameter `G` in Figure 6.7. The class `DenseGraph` is an adjacency-matrix representation of a graph that implements the interface `Graph`. `DenseGraph` is also declared as a generic class that declares a type parameter `C`. This parameter is a place where the layer (that `DenseGraph` will be stored in) will be stored. This parameterization is necessary, because inside `DenseGraph` we would like to access other members declared in the layer, such as `C.V`. Note that we use the `ThisType` construct to prepare for the future extension of this class.

```
interface Graph<&C <: GraphLayer<C.G,C.V>> {
  public ThisType.C.V getVertex(String name);
  public Vector<ThisType.C.V> getChildren(ThisType.C.V v);
  public void addVertex(ThisType.C.V v);
  public void addEdge(ThisType.C.V v1, ThisType.C.V v2);
}

class DenseGraph<&C <: GraphLayer<C.G,C.V>>
      implements Graph<C> {
  private boolean[][] graphArray = new boolean[..][..];
  private ThisType.C.V[] vertexMap = new ThisType.C.V[..];
  ...

  public ThisType.C.V getVertex(String name) { ... }
  public Vector<ThisType.C.V> getChildren(ThisType.C.V v) {
    ... }
  public void addVertex(ThisType.C.V v) { ... }
  public void addEdge(ThisType.C.V v1, ThisType.C.V v2) {
    ... }
}
```

Figure 6.8: A dense graph module

We may also implement a vertex module (e.g. `ColoredVertex`) in the same way. Once these definitions are made, we instantiate the generic layer (we also introduce a `typedef` construct to name the *fixed point* for the type parameters):

```
typedef GraphLayerF =
  GraphLayer<DenseGraph<GraphLayerF>,
             ColoredVertex<GraphLayerF>>;
```

We put the fixed point type, `GraphLayerF`, as an argument for `DenseGraph` and `ColoredVertex`, which are also passed to `GraphLayer`.

Implementing a search layer. `GraphLayerF` can be composed with other layers. `SearchLayer`, shown in Figure 6.9, is one of such layers. Its definition

```
mixin SearchLayer<&G <: SearchG<SearchLayer<G,V>>
                  &V <: SearchV<SearchLayer<G,V>>> {
    ...
}
```

Figure 6.9: A generic traversal layer

```
interface SearchG<&C <: SearchLayer<C.G,C.V>> {
  public void visit(ThisType.C.V v);
}

mixin DFSGraph<&C <: SearchLayer<C.G,C.V>> requires {
        Vector<ThisType.C.V> getChildren(ThisType.C.V v);
      } implements SearchG<C> {

  public void visit(ThisType.C.V v) { ... }
}
```

Figure 6.10: A depth first visitor module

is similar to that of the graph layer in Figure 6.7, except that `SearchLayer` is declared as a mixin.

Figure 6.10 shows an implementation of depth first search module (`DFSGraph`) that will be bound to the type parameter `G` in `SearchLayer`. `DFSGraph` is also declared as a mixin. To enable separate typechecking, it declares an interface that `DFSGraph` requires. In its `requires` clause, it declares the `getChildren` method whose a formal parameter type is `ThisType.C.V` and whose return type is `Vector<ThisType.C.V>`. Since `ThisType` refers to the type of `this` obtained after composition, we may safely compose `DFSGraph` with `DenseGraph` (see below).

As in the case of `GraphLayerF`, we instantiate the generic search layer:

```
typedef SearchLayerF =
  SearchLayer<DFSGraph<SearchLayerF>,
              FlagVertex<SearchLayerF>>;
```

Composing layers. So far, we have developed mixin layers that implement the graph layer and the search layer. Finally, we compose these layers:

```
SearchLayerF::GraphLayerF.G graph =
  new SearchLayerF::GraphLayerF.G();
...
SearchLayerF::GraphLayerF.V vertex = graph.getVertex("..");
graph.visit(vertex);
```

In the composition `SearchLayerF::GraphLayerF`, the same names of "inner classes" appear in each layer. These inner classes are eventually composed when the layers are composed.

6.4 Discussion

Does the property of type soundness still holds when we extend McJava with generics and `ThisType`? Unfortunately, the answer is *no*. We use the following code fragment to show how this property is broken:

```
SearchLayerF::GraphLayerF.V v =
  new SearchLayerF::GraphLayerF.V();
GraphLayerF.V vx = v;
vx.graph = new Foo::GraphLayerF.G();
v.graph.visit(..); // run-time error!!
```

An instance `v` has a type `SearchLayerF::GraphLayerF.V`, and it is assigned to a variable `vx`. Assuming that an `&` annotated type variable accessed via a composition is a subtype of an `&` annotated type variable accessed via a constituent of that composition (covariant subtyping), this assignment is legal. Then, we assign a new value to the instance variable `graph` of `vx` (Assume that `graph` is declared with a type `ThisType.C.G` on `ColoredVertex`. As the statically known type of `vx.graph` is `GraphLayer.G`, and with the covariant subtyping explained above, this assignment is also legal. However, the actual type of `vx.graph` is `SearchLayerF::GraphLayerF.G`; therefore, this assignment actually results in a run-time error!

The cause of this error is a subtype relation between `GraphLayerF.V` and `SearchLayerF::GraphLayerF.V`. Intuitively, this subtyping is derived from the subtype relation between the actual types stored in the type variables (that are `DenseGraph` and `DFSGraph::DenseGraph`). However, when we access them from the outside of layers, there seems to be no obvious relationship between `V`'s at *different levels* of composition; in other words, there may exist no situation when the assignment `GraphLayerF.V vx = v` in the above code fragment becomes necessary. Therefore, we may omit this covariant subtyping, which makes the above code fragment not well-typed:

```
SearchLayerF::GraphLayerF.V v =
  new SearchLayerF::GraphLayerF.V();
GraphLayerF.V vx = v; // compile error
vx.graph = new Foo::GraphLayerF.G();
v.graph.visit(..);
```

Another aspect of our approach that should be discussed here is its complexity. In fact, our approach sacrifices readability of programs in favor of reusability and modularity. The reason why our approach produces complex programs is that, in our approach, the "inner classes" must be parametrized over the layer that is also parametrized. In the original mixin layers, this parameterization is not necessary, because inner classes are contained inside the layer so that they can refer to each other without going through type parameters.

We may enhance understanding of programs written by our method by accumulating experiences of using it. Incorporating with other language features will also reduce this complexity (e.g. [25, 44, 67]).

6.5 Summary

In this chapter, we have discussed relationship between mixins and other language constructs such as the mechanism of generics and `ThisType`. We have shown a complicated issue on mixin-based composition including type parameters, and a natural extension of `ThisType` to mixin-based composition. We have also shown that the language proposed in this chapter (McJava extension with generics and `ThisType`, plus other syntax sugars such as `&` annotated type parameters and `typedef`) has a very strong expressive power, even though the resulting code becomes somewhat tricky.

Chapter 7

Related Work

7.1 Mixin-Based Systems

7.1.1 Jam: Another Approach to Java with Mixins

Jam [4] is an extension of Java with mixins like McJava. Unlike McJava, Jam gives semantics of mixin compositions by translation to Java that is informally expressed as the *copy principle*:

> A class obtained composing a mixin `M` with a class `P` should have the same behavior as a usual subclass of `P` whose body contains a copy of all the elements defined in `M`.

This semantics looks natural, since the obtained composition is exactly the same as a hand-written subclass of `P` whose body is the same as that of `M`. Like McJava, Jam also provides the feature of mixin-types, which means a mixin can be used as a type and a mixin composition is a subtype of both the mixin and the parent class from which it has been composed.

In this scheme, the expression `this` used in a mixin `M` should have the static type `M`. Unfortunately, with this copy principle, there are situations in which the static type of expression `this` inside mixins cannot be determined correctly. One example of such situations is shown in Figure 7.1. In Figure 7.1, class `A` has two overloaded methods `f` with argument types mixin `M` and its composition `H`, respectively. Mixin `M` has a method `g`. Inside `g`, method `f`

```
class A {
  int f(M m) { ... }
  boolean f(H h) { ... }
}
mixin M {
  void g() {
    int i = new A().f(this);
    ...
  }
}
// Jam's syntax for mixin compositions
class H = M extends C {}
```

Figure 7.1: An example of faulty Jam code

is invoked with argument `this`. Since `this` has type `M`, the type of the return value of this method invocation is statically determined as `int`.

Now, let's consider the semantics of composition `H`. By the copy principle, the semantics of composition `H` is equivalent to the following class definition:

```
class H extends C {
  void g() {
    int i = new A().f(this) // error!
    ...
  }
}
```

Inside that definition, the static type of `this` becomes `H`; therefore, the method invocation `new A().f(this)` has static type `boolean`, resulting in an invalid assignment to integer variable `i`.

To avoid this situation, Jam takes a drastic decision to forbid the use of `this` as an argument in method and constructor invocations inside a mixin. We believe that this design decision is not adequate, since recursive reference through `this` is a primary characteristic of object-oriented programming. Most object-oriented languages actually support a more general form of recursion, known as *open recursion*, or *late-binding* of `this` [49]. In Java, for example, we can write an abstract class inside which the expression `this` is used as an argument to method invocation, but the actual binding of `this` is an instance of its subclass that implements all the abstract methods. Similarly, we should be able to write an *abstract subclass* inside which the expression `this` is used as an argument to method invocation, but the actual binding of `this` is deferred and to an instance of a concrete class that implements all the abstract methods. Using `this` as an argument for method invocations in mixin declarations was useful when we implemented an integrated system, which was discussed in section 2.6.

Unlike Jam, McJava does not adopt the copy principle semantics. In McJava, the static type of `this` in mixin `M` is always `M`, even when the mixin `M` is composed with class `C`. At run-time, `this` in `M` is bound to an instance of a composition; e.g., when the expression `new M::C().g()` is executed, `this` is bound to an instance of `M::C`. Which `f` to be invoked is determined at a compile time, that is `int f(M m)`, and since `M::C` is a subtype of `M`, no run-time error occurs at run-time.

Unlike Jam whose semantics is given by translation to Java thus eventually runnable on the standard JVM, McJava's semantics is given at a more abstract level; therefore, we also have to consider how to compile McJava programs. We have also developed a compilation method from McJava to Java.

7.1.2 Other Mixin-Related Systems

Another approach of developing a mixin is to parameterize a superclass of generic classes using type parameters. We have seen an instance of this approach in Chapter 6 that uses templates of C++ [56]. Even though a generic class in Java5 does not support parameterization of its superclass, some extensions of Java with generics allow it [1, 2].

One of the limitations of McJava that is not shared with generic class approach is its disability to express the mixin's superclass type inside the mixin as shown below:

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

However, we may partially solve this problem by adopting a coding convention to make the classes composed with the mixin explicitly implement 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. If this approach is adopted to McJava, the following code

```
mixin Color requires WidgetI {
  WidgetI f;
  ...
}
```

would work for many purposes. There is a design 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 implementation of possible superclasses. Imposing these classes to be a nominal subtype of the 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 `requires` interfaces, 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.[7]. Instead, we take a nominal approach, because most modern object-oriented languages are nominally typed.

To our knowledge, a core calculus for mixin types extending Java was originally developed by Flatt et al.[29]. The novel feature of this calculus, named MixedJava, is its ability to support hygienic mixins (also founds in [2, 42]). 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 exist two implementations of hygienic mixins [2, 42], neither of which conforms with the McJava type system; 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 to our McJava compiler has been discussed in Chapter 5.

Mixin modules [22], essentially motivated by the problem of interaction with recursive constructs that cross module boundaries in module systems of functional languages, mainly focus on facilitating reuse of large scale programming constructs such as frameworks [23]. Our work, on the other hand, mainly focuses on integrating mixin-types and its flexible subtyping with real programming languages. The work [23] sacrifices mixin subtyping in favor of allowing method renaming.

MixJuice [33] is also independently proposed by Ichisugi et al. to modularize large scale compilation unit. MixJuice is designed as an extension of Java with *difference-based modules* that are separately compilable units of encapsulation. The design of mixins in MixJuice is different from our work. In MixJuice, the *providers* of mixins control encapsulation. In the case of diamond inheritance, the users have the responsibility of composing them without breaking encapsulation. In McJava, on the contrary, the *users* of mixins control encapsulation because these mixins are parametrized over their superclasses. Users add superclasses to mixins and there are no case of diamond inheritance.

Schärli et al. proposed *traits* [53], 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.

7.2 Method Combination in Object-Oriented Languages

As mentioned earlier, our approach of selective method combination is an extension of hygienic mixins [2, 42]. As discussed above, the implementation of hygienic mixins is based on MixedJava. As mentioned earlier, MixedJava uses run-time context information 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 the run-time inheritance chain to be different from the statically known superclass. Selective call of the "original" method to `super` is not achieved in [2, 42, 29].

Ernst proposed the *propagation* mechanism of method combination in the statically typed language `gbeta` [26], a generalization of the language BETA [41]. `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. Since `gbeta` does not allow intentional overriding that is allowed in McJava, propagation mechanism in `gbeta` is simpler than selective method combination in McJava. There is a design tradeoff between which approaches to take, `INNER` or `super`; further discussion about this tradeoff is found in [9]. We also note that recently Goldberg et al. propose a language that integrates `super` and

`INNER` [31].

7.3 Other Related Issues

Aspect-oriented programming (AOP) [39] aims to modularize cross-cutting concerns in modules called *aspects*. Some kinds of cross-cutting concerns are also modularized using mixins. We have already shown an instance of this modularization in section 2.6. In this sense, McJava weakly supports AOP but some additional efforts are required to programmers. Especially, we need to write a glue code composing mixins instead of using a *weaver*. However, we may note that constructing "aspects" by using mixin-based composition becomes easier if we adopt the layered design discussed in Chapter 6.

In Chapter 6, we have proposed a McJava extension for mixin layers. Cardone et al. also proposed a Java extension for mixin layers named JL [17]. Unlike mixin layers, JL supports `ThisType`. Like mixin layers, in JL the members in a layer are expressed as inner classes. Our approach, on the other hand, enables separation of these inner elements from the layer.

Another idea explained in Chapter 6 is the introduction of `&` annotated type parameters, which allows access to each type parameter from the outside of generic class; in other words, if a type parameter is declared with `&` annotation, it may also be treated as a field of this class. This mechanism is a combination of the benefits of virtual types and generic classes. The similar system is also proposed by Thorup et al. [62]. However, our approach is slightly different from virtual types in that `&` annotated type parameters are treated as fields declared in *each concrete class obtained by assigning types to the type parameters* (note that it is different from class variables), while virtual types are instance variables. Note that virtual types cannot be accessed through types, like `C.V` that is required in the example program shown in Chapter 6.

Our approach discussed in Chapter 6 is a generalization of extensible mutually recursive types [15], in that in our approach each extension is also parameterized over the extended (original) module. A feature of extending mutually recursive types "at once" is considered to be promising way to solve some challenging issues of object-oriented programming such as *expression problem* [12]. There are also much work to this direction of research (e.g., *family poly-*

morphism [24], *higher order hierarchies* [25], and *nested inheritance* [44]). The programming language Scala also provides such extensibility [46].

Mixins may be used as vehicles to directly implement *roles* in terms of role modeling [59]. Epsilon [65, 60], 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 good basis for incorporating static typing into Epsilon. When an Epsilon object dynamically adapts 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 dissertation such as selective method combination provides a good basis for incorporating similar mechanism into Epsilon.

Chapter 8

Conclusion

8.1 Summary of the Dissertation

In this dissertation, we have studied a mechanism of mixin-based composition in the context of Java-like languages both on theoretical point of views and implementation point of views. We have designed and implemented a programming language McJava, an extension of Java with mixins. We have also studied more advanced aspects of mixin-based composition such as selective method combination and interaction with other language constructs. The main contributions are summarized as follows:

- We have designed McJava that extends Java with new syntactic forms such as mixin declarations and mixin composition operators. This language has an ability to use the name of a mixin as a type. The language also supports more advanced features such as higher order mixins and mixin-based subtyping. The example of integrated systems has illustrated the expressive power of McJava.

- We have developed Core McJava, a small calculus of McJava, and a proof of the type soundness theorem of Core McJava. Core McJava includes key constructs that characterize McJava type system, ensuring the soundness of McJava type system. Core McJava also includes the feature of method overloading without suffering the problem faced by Jam.

- We have studied an implementation strategy of McJava compiler. This mechanism ensures that McJava programs are runnable on any standard Java virtual machines and McJava does not degrade run-time performance of Java. With this compilation, it is also guaranteed that the existing Java libraries can be used in McJava without any changes to the libraries.

- 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 appropriately select a method to be called. This approach promotes flexibility of mixin-based compositions, and reliability of programs, because our approach makes it easier to preserve the behavior of classes. We have implemented the mechanism into the McJava compiler.
- We have designed an extension of McJava with generics and `ThisType`. We have discovered the language is not type-sound, but we can recover type soundness by imposing restrictions on covariant subtyping among inner mixins. We have also shown the expressive power of the language that allows the design of generic mixin layers.

In short, this dissertation provides a convincing way for adding mixins into Java. We may also say that the similar approach may be applied to nominally typed object-oriented languages other than Java such as C#, because our model does not include any "only Java-specific" features.

8.2 Future Work

Future work mainly consists of two directions: modeling of features left out from the dissertation, and more practical implementation of McJava.

Modeling Other Aspects of Our Work. We have informally presented the McJava compilation strategy. Using a formal method will enhance understanding of the correctness of this compilation. A possible way for doing this is to formalize a target language of compilation (that will be a core of Java that includes interfaces because McJava compilation strongly depends on the existence of interfaces in the target language), then to formalize translation from Core McJava to the target language.

One significant aspect that Core McJava does not include is selective method combination. To include it, we have to extend Core McJava dynamic semantics with the ability of referring the static type of a receiver of method invocation during the reduction process. After this extension, a more careful study on compilation of selective method combination will be required to reduce the complexity of the current implementation.

The presentation of the design of an extension of McJava with generics and `ThisType` is also informal. We may also have to add these features to Core McJava to formally discuss on the properties of the language.

More Practical Implementation. Current implementation of McJava compiler is experimental. That means the purpose of the implementation is only to experiment that the implementation mechanism is correct. For more practical use, we should satisfy the usability requirements of the compiler such as compilation into byte code, not into source code of Java, and separate compilation support for mixins. Fortunately, there are some projects of developing extensible Java compilers [45]. We may use such products as a basis for more practical implementation of McJava compiler.

Bibliography

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Conference Proceedings of OOPSLA '97, Atlanta*, pages 49–65. ACM, 1997.

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

[5] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.

[6] Lorenzo Bettini, Viviana Bono, and Silvia Likavec. A calculus of mixin-based incomplete objects. In *The Eleventh International Workshop on Foundations of Object Oriented Languages (FOOL 11)*, 2004.

[7] 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.

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

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

[10] 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.

[11] Gilad Bracha, Martin Odersky, David Stroutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA 1998*, pages 183–200, 1998.

[12] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 82(8), 2003.

[13] Kim B. Bruce, Adran Fiech, Angela Schuett, and Robert van Cent. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP '95*, volume 952 of *LNCS*, pages 27–51, 1995.

[14] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 389–413, 2004.

[15] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, volume 1445 of *LNCS*, pages 523–549, 1998.

[16] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[17] Richard Cardone, Adam Brown, Sean McDirmid, and Calvin Lin. Using mixins to build flexible widgets. In *AOSD 2002*, pages 76–85, 2002.

[18] Richard Cardone and Calvin Lin. Comparing frameworks and layered refinement. In *ICSE 2001*, pages 285–294, 2001.

[19] Robert Cartwright and Jr. Guy L. Steele. Compatible genericity with run-time types for the Java programming language. In *OOPSLA 1998*, pages 201–215, 1998.

[20] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for

Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 130–145, 2000.

[21] Patrick Donohoe, editor. *Software Product Lines: Experience and Research Directions: Proceedings of the First Software Product Lines Conference (SPLC1)*. Kluwer Academic, 2000.

[22] Dominic Duggan and Constantinous Sourelis. Mixin modules. In *ICFP'96*, pages 262–272, 1996.

[23] Dominic Duggan and Ching-Ching Techaubol. Modular mixin-based inheritance for application frameworks. In *OOPSLA 2001*, pages 223–240, 2001.

[24] Eric Ernst. Family polymorphism. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 303–327, 2001.

[25] Eric Ernst. Higher-order hierarchies. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 303–329, 2003.

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

[27] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of ICFP 1998*, pages 98–104, 1998.

[28] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 365–389, 2004.

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

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

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

[32] Ian M. Holland. Specifying reusable components using contracts. In *ECOOP 1992*, volume 615 of *LNCS*, pages 287–308, 1992.

[33] Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism. In *Proceedings of ECOOP 2002*, pages 62–88, 2002.

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

[35] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2003.

[36] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. In *ECOOP 2002*, volume 2374 of *LNCS*, pages 441–469, 2002.

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

[38] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[39] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings ECOOP'97*, pages 220–242, 1997.

[40] 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.

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

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

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

[44] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA 2004*, pages 99–115, 2004.

[45] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152, 2003.

[46] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003*, LNCS, 2003.

[47] Martin Odersky and Philip Wadler. Pizza into Java: Translation theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 146–159, 1997.

[48] David L. Parnas. Designing software for ease of extension and contraction. In *ICSE'78*, pages 264–277, 1978.

[49] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[50] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11*, pages 297–306, 2003.

[51] Didier Rémy and Jérôme Vouillon. The reality of virtual types for free! http://cristal.inria.fr/ remy/publications.html, October 1998. Unpublished note avaliable electronically.

[52] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 16–25, 2004.

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

[54] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002.

[55] Alan Snyder. Inheritance and the development of encapsulated software systems. In *Research directions in object-oriented programming*, pages 165–188. MIT Press, 1987.

[56] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[57] Kevin Sullivan, Lin Gu, and Yuanfang Cai. Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ. In *Proceedings of 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 19–26, 2002.

[58] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transaction on Software Engineering and Methodology*, 1(3):229–268, 1992.

[59] Tetsuo Tamai. Objects and roles: modeling based on the dualistic view. *Information and Software Technology*, 41(14):1005–1010, 1999.

[60] 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.

[61] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP 1997*, volume 1241 of *LNCS*, pages 444–471, 1997.

[62] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *ECOOP'99*, volume 1628 of *LNCS*, pages 186–204, 1999.

[63] Mads Torgersen. The expression problem revisited – four new solutions using generics. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 123–143, 2004.

[64] Mads Torgersen, Christian Plesner Hansen, Erik Ernst Peter von der Ahe, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Symposium on Applied Computing (SAC 2004)*, pages 1289–1296, 2004.

[65] 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.

[66] 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.

[67] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. I&C Technical Report 200433, EPFL, March 2004.