Mostly Modular Compilation of Crosscutting Concerns by Contextual Predicate Dispatch

Shigeru Chiba
Tokyo Institute of Technology, Japan
www.csg.is.titech.ac.jp

Atsushi Igarashi
Kyoto University, Japan
www.sato.kuis.kyoto-u.ac.jp

Salikh Zakirov
Tokyo Institute of Technology, Japan
www.csg.is.titech.ac.jp

Abstract

The modularity of aspect-oriented programming (AOP) has been a controversial issue. To investigate this issue compared with object-oriented programming (OOP), we propose a simple language providing AOP mechanisms, which are enhanced traditional OOP mechanisms. We also present its formal system and then show that programs in this language can be only *mostly* modularly (*i.e.* separately) typechecked and compiled. We mention a source of this unmodularity and discuss whether or not it is appropriate to claim that AOP breaks modularity compared with OOP.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features — Classes and objects

General Terms Languages

Keywords Aspect Oriented Programming, Java, AspectJ.

1. Introduction

Aspect-oriented programming (AOP) has been actively studied for more than a decade. A challenge of AOP is to manage crosscutting concerns in a modular fashion. What is a crosscutting concern? According to Kiczales et al. [25], "In general, whenever two properties being programmed must compose differently and yet be coordinated, we say that they cross-cut each other." A crosscutting concern consists of such crosscutting properties. Hence, by definition, it might seem anti-modular; something not locally complete or self-contained at all. This might give us an impression that AOP is a paradigm to "*modularize the un-modularizable*" [29] but this phrase sounds contradictory or at least confusing.

Typical AOP languages such as AspectJ [26] increase this confusion and, what is worse, they have lead some developers to feeling that AOP languages do not really modularize programs but rather break modularity in a classical sense. As we show in this paper, AspectJ programs cannot be modularly (*i.e.* separately) typechecked or compiled. This fact would not be inherent to AOP in general but it would be an intentional design decision for maximizing practical usefulness of the language. However, some users claimed that "*Given that AOP has set out to modularize crosscutting concerns (its methodological claim), but by its very nature (its mechanics) breaks modularity*" [42]. To respond such confusion, Kiczales and Mezini proposed a new interpretation of modularity [27], in which a module interface is context dependent on a deployment configuration of classes and aspects. In fact, a module interface of AspectJ is not determined in their sense until a list of all woven aspects is given. The authors of [27] also mentioned "*crosscutting concerns inherently require global knowledge to support reasoning*." The need of global analysis is, however, a sign of being unmodular.

To investigate this confusion, we attempt to develop a simple Java-like AOP language whose programs can be modularly typechecked and compiled without a global analysis. This language is named *GluonJ*. This paper reports this attempt. This language supports a useful subset of the AOP functionality of AspectJ. If a program in this language can be modularly compiled, then it will be revealed that unmodularity is not inherent to AOP or crosscutting concerns in general. Even if it cannot, we will see the source of the unmodularity. Being modularly compilable does not mean modular reasoning is also possible. However, it will be a solid basis of discussion. Directly discussing modular reasoning is difficult. For example, it is not clear that modular reasoning is possible in object-oriented programming (OOP) because of method overriding.

To ease comparison with traditional modular OOP languages, the AOP mechanisms of our language are simple enhancement of method overriding and method dispatch of OOP. They are not based on the popular pointcut-advice of AspectJ. The pointcut-advice is too different from OOP mechanisms to be compared side-by-side with respect to modularity. Our language adopts the idea of predicate dis-

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

patch [16, 32], which is modified to use external calling contexts, such as *who is a caller*, so that method dispatch can be used for AOP. This modified mechanism is called *contextual predicate dispatch*.

Unfortunately, it turns out that a program written in GluonJ cannot be completely modularly typechecked or compiled. GluonJ is only mostly modular and a limited global analysis is necessary. However, it is not inherent to the basic idea of the language; accepting the global analysis is our design decision. We can avoid the global analysis by changing the language design of GluonJ if we can degrade its practical usefulness to a certain degree. Furthermore, our compilation technique for minimizing runtime overheads due to AOP also needs global analysis and transformation at link time or load time. This paper, however, shows that only limited global transformation is necessary.

Our contribution is twofold. (1) We propose a language providing AOP mechanisms, which are not traditional ones but rather enhanced OOP mechanisms for dealing with crosscutting concerns. This makes it easy to investigate the modularity of AOP mechanisms by comparing with well-known OOP mechanisms. (2) We show programs in that language are mostly modularly compiled although it is often criticized that AOP will break modularity. Then we discuss the modularity of AOP.

In the rest, Section 2 briefly overviews the AOP functionality discussed in this paper. Section 3 proposes our language GluonJ, which provides AOP functionality in the OOP style. Sections 4 to 6 show the calculus and implementation of GluonJ. Section 7 mentions related work and Section 8 concludes this paper.

2. Language mechanisms for AOP

This section briefly presents the AOP mechanisms that we deal with in this paper. Then it shows that AspectJ programs using those mechanisms are not modularly compiled.

2.1 Destructive extension

A combination of execution pointcut and advice is one of the most useful mechanisms of AspectJ. It enables *destructively* modifying an existing method. The source code of that method does not change due to the obliviousness property of AOP [17].

Figure 1 shows a simplified example of interpreter written with aspects. The Calc class has a main method, which makes an instance of Parser to parse a program given as a command-line argument. Then it evaluates the program by calling the eval method. Suppose that this language can compute only an integer value. We can extend this program by writing an aspect to support floating-point numbers besides integers. For example, the AddExpr class, which is for the + nodes of an abstract syntax tree, originally can process only integer values but it can be extended by the FloatExt aspect to compute floating-point numbers as well. Its around advice

```
public class Calc {
  public static void main(String[] args) {
    ASTree t = new Parser().parse(args[0]);
    t.eval();
  }
}

class AddExpr extends ASTree {
  Value eval() {
    return new IntValue(left.eval().intValue()
           + right.eval().intValue());
  }
}

public aspect FloatExt {
  Value around(AddExpr ae):
      execution(Value AddExpr.eval()) && this(ae) {
    if (ae.left.isType(Integer.class)
        && ae.right.isType(Integer.class))
      return proceed();
    else
      return new DoubleValue(ae.left.eval().doubleValue()
                          + ae.left.eval().doubleValue());
  }
}
```

Figure 1. A simple interpreter in AspectJ.

substitutes for the original eval method in AddExpr. This is similar to method overriding. proceed() invokes the original method body of eval.

Note that the aspect *destructively* modifies the behavior of the eval method in AddExpr and hence we do not have to modify the parse method in the Parser class to instantiate a new version of the AddExpr class when it finds a + expression.[1] The original program (Calc and AddExpr) remains the same; the behavior reverts to the original if the FloatExt aspect is removed.

If the compiler above is implemented without AOP, this level of customizability is not available by other extension mechanisms such as inheritance, mixins [7], mixin layers [40], family polymorphism [15], traits [39], and nested inheritance [34]. These mechanisms let an extended version of a class, namely a subclass, coexist with the original version of that class. Thus, if programmers want to use the new behavior for some instances, they must explicitly declare that fact, for example, by giving the subclass name when the instances are created. The other instances remain showing the old behavior. This is beneficial when reusing *parts* of a program, such as classes, since both a new version and its original version can be used in the same program. For example, programmers can implement a JarFile class as a subclass of ZipFile class and use both in one program. On the other hand, when programmers want to use only the new behavior, they have to modify all occurrences of object creation included in the original program to use the new version.

[1] No matter whether we use AOP, we must modify the Parser class to parse a floating-point constant.

AOP does not allow original and new versions of a class to coexist in a program. It makes only the new version available in the program. This *destructive* feature of AOP is useful when programmers want to reuse a *whole* program and partly customize it to build new software. Reusing a whole program has practical needs. A good example is JastAdd [14], which is a compiler construction framework. Since JastAdd contains a simple AOP mechanism, programmers can develop a Java 1.5 compiler by only writing aspects for extending a Java 1.4 compiler. They do not have to modify object-creation expressions in the original program of the Java 1.4 compiler. All extensions are separated into the aspects. We hence easily switch Java 1.4 and 1.5 by adding/removing the aspects.

2.2 Limited scope

In AspectJ, the within/withincode pointcut enables to apply an extension into a limited scope. In particular, a combination of call and withincode pointcuts enables to execute an advice *in the middle of* a method body. This partial extension of a method body is a unique feature of AOP. Note that we do not have to modify the source code of the method body. Since it may be considered as breaking the information hiding principle [37] on a method, its benefits have been controversial [42] and several ideas for dealing with this have been proposed so far [1, 19]. On the other hand, this feature is necessary to implement a number of crosscutting concerns represented by a famous logging concern.

Figure 2 presents a simple Logging aspect. Its before advice is invoked in the middle of the eval method in the VariableDecl class, namely, just before the recordVariable method in Env is called from the eval method. The call pointcut selects all occurrences of recordVariable calls and the withincode selects only the occurrences in the eval method.

Although the Logging aspect is a simple example, the same idea can be used to implement a variety of crosscutting concerns including performance profiling, transaction, synchronization, and security. For example, it is possible to write around advice for performing mutual exclusion to make the method thread-safe when the recordVariable method in Figure 2 is executed.

2.3 Modular compilation

The AspectJ designers do not seem to have prioritized modular typechecking. For example, a proceed call in an around advice is not type-safe due to its generic nature. This problem was solved by StrongAspectJ [18]. Furthermore, an early AspectJ compiler needs to compile all source files composing a program together. Although a later compiler can modularly compile each source file and finally link all the compiled binary files, programmers still need to manually give a compiler some source files together. For example, if a method calls another method introduced by an aspect, that aspect must be compiled together with the caller method. In Figure 3, the print method in the Printer aspect

```
public class VariableDecl extends ASTree {
  String name;
  Value eval() {
    Value initValue = right.eval();
    env.recordVariable(name, initValue);
    return null;
  }
}

public aspect Logging {
  before():
    call(void Env.recordVariable(String,Value))
    && withincode(Value VariableDecl.eval()) {
    System.out.println("declare a variable");
  }
}
```

Figure 2. A logging aspect.

```
public aspect Printer {
  public void AddExpr.print() {
    System.out.println(this.opName());
  }
}

public aspect OperatorName {
  public String AddExpr.opName() {
    return "+";
  }
}
```

Figure 3. An intertype declaration depending on another

calls the opName method introduced by the OperatorName aspect. If only the AddExpr class and the Printer aspect are compiled without the OperatorName aspect, the compiler will report opName is not found since it cannot see the OperatorName aspect.

The lack of modular typechecking might be not intrinsic in AOP in general. It might be an intentional design decision for practical flexibility and fixing this problem might be possible by adding extra declarations. However, this confuses the discussion on the modularity of AOP. If modular typechecking were possible, the checkable properties would be the base of the discussion on the modularity.

3. GluonJ

This section presents our simple AOP language named *GluonJ*, which provides the AOP functionality mentioned in the previous section. GluonJ is an extension of Java and introduces a new language construct called *reviser* into Java.

3.1 Reviser

A reviser is similar to a class but the member declarations of a reviser are considered as part of the definition of its target class. They are merged into its target's. Figure 4 is an example of reviser. Like open classes [12], this reviser Printing directly appends new members, a tag field and a print method, to its target class AddExpr. A reviser cannot be instantiated

```
class Printing revises AddExpr {
  String tag = "+";
  void print() {
    System.out.println(left + tag + right);
  }
}
```

Figure 4. A reviser adding new members.

```
class FloatExt revises AddExpr {
  Value eval() {
    if (left.isType(Integer.class)
        && right.isType(Integer.class))
      return super.eval();
    else
      return new DoubleValue(left.eval().doubleValue()
                             + right.eval().doubleValue());
  }
}
```

Figure 5. A reviser equivalent to the aspect in Figure 1.

like an abstract class of Java, which cannot be instantiated either. Hence a reviser cannot declare a constructor. Note that the initial value of tag is given in the declaration; no explicit constructor can be declared.

If a method name in a reviser is the same as a method in its target class, then the method implementation in the reviser destructively replaces (or *overrides*) the original implementation in the target class as an around advice with an execution pointcut does in AspectJ. Figure 5 presents a reviser that performs the same modification that the FloatExt aspect in Figure 1 does. Now a call to eval on an AddExpr object invokes the implementation described in the FloatExt class. A call super.eval() included in FloatExt invokes the original implementation of the eval method in AddExpr.

Unlike a method, a reviser cannot revise a constructor in its target class since a reviser cannot declare an explicit constructor. Eliminating this restriction would not be difficult but it is one of future possible extensions of the language.

A static method in a reviser may substitute for the corresponding static method implementation declared in its target class. Suppose that the AddExpr class declares a static method foo and the reviser FloatEx also declares foo. Then, a call AddExpr.foo() invokes the implementation of foo in the reviser FloatEx. Within the body of the foo method in the reviser, however, a call AddExpr.foo() invokes the implementation by AddExpr while a call FloatEx.foo() invokes the implementation by FloatEx.

3.2 A within method

To apply destructive extension in a limited scope, GluonJ adopts the idea of predicate dispatch. A reviser can declare a method with a predicate. With predicate dispatch, a method implementation may have a predicate as a guard. It overrides another if its predicate is true and implies the other predicate.

```
class Tracer revises Env {
  void recordVariable(String name, Value value)
       within VariableDecl.eval() {
    System.out.println("declare a variable");
    super.recordVariable(name, value);
  }
}
```

Figure 6. A method with a predicate.

In JPred [32], a predicate may be a test of an argument type or value, or the receiver's field value.

In GluonJ, only the within predicate is available to provide the capability of the within/withincode pointcut of AspectJ. We below call a method with this predicate *a within method*. The within predicate becomes true if a method-caller's site is *within* the location given by the argument. Like in AspectJ, with a within method, GluonJ enables to append a method invocation in the middle of the body of another method. Since a predicate referring to external calling contexts is unique to AOP, we call the method dispatch in GluonJ *contextual predicate dispatch*.

Figure 6 shows an example of within method. Note that the method signature is followed by a within clause, which specifies a class or a method. The reviser Tracer adds logging functionality as the Logging aspect in Figure 2 does. The implementation of recordVariable in this reviser is invoked only when the method is called from the eval method in the VariableDecl class. It prints a message and then invokes the original implementation in Env by super.recordVariable(...). On the other hand, when the method is called not from the eval method, the original implementation in Env is invoked since the predicate is false. A compile error would be reported if a program does not provide a method implementation without a predicate or with a predicate matching the caller's site. Note that a reviser can declare a within method that does not override any method implementation in its target class.

3.3 Combining revisers

GluonJ allows multiple revisers to revise the same class. The precedence order among reviser is specified by a requires clause in a class declaration. Figure 7 shows an example. This reviser StringExt declares that it has higher precedence than another reviser FloatExt in Figure 5, which revises the same target AddExpr. It also declares that StringExt runs together with FloatExt. A requires clause can include multiple reviser names separated by comma. The left has higher precedence. Moreover, a reviser may have a requires clause including the name of another reviser that revises a different class.

A requires clause is used to incrementally implement a new reviser by reusing other revisers. A required reviser, which has lower precedence, is applied to the target class before the reviser requiring that. The precedence order is transitive. In a valid GluonJ program, the precedence order

```
class StringExt requires FloatExt revises AddExpr {
  Value eval() {
    if (left.isType(Number.class)
        && right.isType(Number.class))
      return super.eval();
    else
      return new StringValue(left.eval().toString()
             + right.eval().toString());
  }
}
```

Figure 7. A requires clause.

among the revisers revising the same class must be a total order. If not, a compile error will be reported.

A method implementation declared in a reviser may *override* another reviser's implementation if the reviser has higher precedence than the other. Their target class has lower precedence than them. A call on super invokes the overridden implementation in a reviser with lower precedence (or the target class). This is also true for within methods. On a method call, a reviser with higher precedence is searched first to select an implementation matching the calling contexts. The target class is searched last. This rule is significant if two revisers for the same target class have implementations of the same method but with different predicates such as within C and within C.foo(). Suppose that a caller site is the foo method in C. If a reviser with higher precedence has the implementation with within C, it is selected instead of the implementation with within C.foo() although the latter is more specific. The formal semantics of method dispatch is presented in the next section.

3.4 Using a new member

A reviser can refer to members added by other revisers specified by its requires clause. In Figure 7, StringExt can refer to members added by FloatExt (if any). On the other hand, a class needs a using declaration when it refers to members added by a reviser. The added members are visible only within the source file including the using declaration of that adding reviser. Figure 8 shows an example. The Printer class can call the print method on an AddExpr object since the source file includes a using declaration of the reviser Printing.

This restriction might seem to decrease the obliviousness property of GluonJ. The reader might think that a source file must be modified to include a using declaration and hence reusing a whole program *as is* would be made difficult. However, this restriction is not a problem. If an original program is self-contained, it never accesses newly added members since it was written before the reviser. Only the classes written with or after the reviser will access the added members and thus it is acceptable to enforce programmers to include a using declaration in the source file of those classes. On the other hand, a using declaration helps modular type checking.

```
using Printing;    // a using declaration
class Printer {
  static void print(AddExpr e) {
    e.print();
  }
}
```

Figure 8. A class using a method added by a reviser.

3.5 Visibility rule

A reviser follows the same visibility rule as normal classes. It may only access public members, *so-called* package members in the same package, protected members of the target class and its super classes, and private members of the reviser itself. It may also access protected members of other revisers sharing the same target class but having lower precedence. Method overriding by a reviser also follows the rule of the standard Java. It cannot override a private method in the target class. This rule restricts the extensibility by revisers but we adopt this rule for the information hiding principle [37]. This rule will also ease the fragile pointcut problem of AOP [28, 31, 43]. In a *good* Java program, a non-private method can be expected to be available until the design of the program is largely changed due to refactoring.

3.6 Current Limitations

Dynamic pointcuts

Although AspectJ provides *dynamic* pointcuts such as cflow, GluonJ does not deal with them. Introducing other predicates like cflow and improving the AOP capability of GluonJ is a straightforward extension as our previous workshop paper [9] showed. However, our goal is modular typechecking and compilation; the dynamic behavior of dynamic pointcuts obviously complicates type checking, in particular, checking the exhaustiveness property. This property guarantees that "no such method is found" errors never happen during runtime. Unless a method with a dynamic predicate like cflow always overrides a "default" method, which has no dynamic predicate, GluonJ could not statically check that this property is preserved.

This limitation might look serious but the static pointcuts discussed in this paper are still a useful subset. For example, the AspectJ support for Eclipse AJDT 2.0.2 consists of 2212 classes and 47 aspects. It includes only 8 occurrences of cflow while it includes 60 execution, 92 call, and 97 within/withincode pointcuts. A drawing tool AJHotDraw 0.4 includes no cflow although it includes 18 execution, 30 call, and 36 within/withincode in 290 classes and 31 aspects.

Other AspectJ features

For the same reason of modular typechecking, GluonJ does not support the pattern matching feature of AspectJ, with which a pointcut argument may include wild cards and/or enumerate several join points. For example, in GluonJ, a within method cannot override multiple methods whose sig-

natures match a pattern. A single reviser cannot revise multiple classes. Although this pattern matching feature is useful to implement a homogeneous concern, GluonJ does not support it and hence its main applications are heterogeneous concerns.

Although AspectJ provides abstract aspects and pointcuts, GluonJ does not. These enable separating concrete definitions of pointcuts into sub-aspects and therby they improve the reusability of aspects. An example of the use is found in [20]. Introducing this template-like mechanism into GluonJ for better code reusability is out of scope of this paper.

super and proceed

A call on super in a within method invokes the overridden implementation, either in a next reviser with lower precedence or in its target class if there are no more revisers. The search for the target method of the super call uses the reviser as the caller site when checking a within predicate. It does not use the original caller. Suppose that a method foo in C invokes a within method bar in a reviser R. If bar calls super.bar(), then the caller site for the search is not foo in C but bar in R. Thus, the implementation with a predicate within C.foo() is not selected.

This semantics is different from proceed in AspectJ since proceed searches aspects under the same contexts as the original. Extending GluonJ to provide a proceed-like mechanism is our future work. GluonJ adopts the semantics above for keeping super calls simple. Note that a within method bar may call a different method on super, for example, super.bar2() instead of super.bar(). GluonJ searches for both super.bar2() and super.bar() under the same calling contexts. Another design option is that the caller site is bar in R for super.bar2() while it is foo in C for super.bar(). GluonJ, however, does not adopt this design because of its complication.

Multiple within methods

If a reviser declares a method foo with a within predicate, then it cannot declare another method foo with the same signature with/without a predicate. This limitation is just for simplifying the specification and implementation of GluonJ. To declare more than one within method for foo, multiple revisers for the same class must be declared and each of them must declare one within method. As in Java, a reviser can declare multiple methods foo if they have different signatures.

4. Core calculus of GluonJ

In this section, we give a formal calculus called GluonFJ as an extension of Featherweight Java (FJ) [24]. GluonFJ adds revisers and within methods to FJ. For simplicity, we do not model super calls, and fields of revisers; within clauses are restricted so that only class names can be given as source code locations.

The purpose of developing a formal calculus is twofold: (1) to show type soundness of the core language as usual and (2) to rigorously discuss to what extent type checking and compilation are modular. Actually, our proof of type soundness is an easy extension of one for FJ, thanks to the fact that most of the features of GluonJ are extensions of those of traditional object-oriented languages. We discuss in what sense typechecking is (mostly) modular in the next section. Later in Section 6, we give a formal translation from GluonFJ to FJ to model compilation and argue that compilation is also mostly modular.

We first give the syntax, typing rules, and operational semantics of GluonFJ and then state type soundness. Here, we show only a main part of definitions and the statements of theorems here; the full definition of the language and proof sketches are found in the companion technical report [10].

4.1 Syntax

As mentioned above, GluonFJ has revisers and within methods in addition to most of the features of FJ. Although they are important in compilation as we see in the next section, typecasts have been removed from GluonFJ, since they are orthogonal to the additional features.

The syntax of GluonFJ is given as follows:

CL	::=	`class C extends C using` $\overline{\texttt{R}}$ `{` $\overline{\texttt{C}}$ $\overline{\texttt{f}}$`;` $\overline{\texttt{M}}$ `}`	class
	|	`class R revises C using` $\overline{\texttt{R}}$ `{` $\overline{\texttt{M}}$ `}`	reviser
L	::=	C | R	location
M	::=	`C m(`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`){ return e; }` [`within L`]	method
e	::=	x | e.f | e.m($\overline{\texttt{e}}$) | `new C(`$\overline{\texttt{e}}$`)` | e `in L`	expression
v, w	::=	`new C(`$\overline{\texttt{v}}$`)`	value

Following the convention of FJ, we use an overline to denote a sequence and write, for example, $\overline{\texttt{x}}$ as shorthand for $\texttt{x}_1, \ldots, \texttt{x}_n$ and $\overline{\texttt{C}}$ $\overline{\texttt{f}}$`;` for "$\texttt{C}_1$ $\texttt{f}_1$`;`, …, $\texttt{C}_n$ $\texttt{f}_n$`;`". The empty sequence is written $\bullet$. The metavariables B, C, D, and E range over class names; R ranges over reviser names; m ranges over method names; and x and y range over variables, which include the special variable this. For technical convenience, we assume that class names and reviser names are disjoint and the (denumerable) set of reviser names is totally ordered. This total order represents the precedence in method dispatch and so there are no requires clauses in class definitions. In what follows, we assume that any sequence of reviser names $\overline{\texttt{R}}$ is sorted according to this order (the smaller the index is, the higher the precedence is).

CL is a class (or reviser) definition, consisting of its name, a super class name (or the class name that it revises, respectively), reviser names that it uses, fields, and methods. A reviser cannot have fields. We omit explicit constructor definitions, which take initial values of all fields and set them to the corresponding ones. Types of GluonFJ are only class names, hence C for field, parameter, and return types. A method definition M can have an optional clause within L, where

L, standing for locations, is a class/reviser name. One class cannot have more than one method of the same name. The body of a method is a single `return` statement, following FJ. Expressions are mostly the same as FJ except for omitted typecasts and the new form `e in L`, which is used to mark which class `e` originates from in the operational semantics. This form is not supposed to appear in class definitions. We will denote a substitution of expressions $\overline{\mathtt{e}}$ for variables $\overline{\mathtt{x}}$ by $[\overline{\mathtt{e}}/\overline{\mathtt{x}}]$.

A GluonFJ program is a triple consisting of a class table *CT*, which is a mapping from class names to class definitions, a reviser table *RT*, which is also a mapping from reviser names to reviser definitions, and an expression, which stands for the body of the main method. We write $dom(CT)$ (and $dom(RT)$) for the domain of the table and write `C ext D` when $CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \cdots}$. Similarly, we use `R rev C` and `L using` $\overline{\mathtt{R}}$.

Finally, we always assume fixed class tables, which are assumed to satisfy the following sanity conditions: (1) $CT(\mathtt{C}) = \mathtt{class\ C\ \cdots}$ for every $\mathtt{C} \in dom(CT)$ and similarly for *RT*; (2) $\mathtt{Object} \notin dom(CT) \cup dom(RT)$; (3) for every name L (except `Object`) appearing anywhere in *CT* and *RT*, we have $\mathtt{L} \in dom(CT) \cup dom(RT)$; and (4) there are no cycles formed by `extends` clauses.

4.2 Lookup functions

As in FJ, we need functions to look up fields, method signatures and bodies in the class tables. The function $fields(\mathtt{C})$ returns all the fields of C and its super classes with their types as $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$. We omit its straightforward definition (which in fact is the same as in FJ). The function $fstrev(\mathtt{C}, \overline{\mathtt{R}})$, whose definition is omitted, returns the first reviser that revises C, found in $\overline{\mathtt{R}}$, or just returns C if there is no such reviser. The function $next(\mathtt{L}, \overline{\mathtt{R}})$ returns the next class of L to look up and is defined by:

$$
\begin{array}{rcl}
next(\mathtt{R}_i, \overline{\mathtt{R}}) & = & \begin{cases} \mathtt{R}_j & \text{if } j > i, \mathtt{R}_i\ \mathtt{rev\ C}, \mathtt{R}_j\ \mathtt{rev\ C}, \\ & \text{and } \neg\exists k \in (i,j).\mathtt{R}_k\ \mathtt{rev\ C} \\ \mathtt{C} & \text{if } \mathtt{R}_i\ \mathtt{rev\ C} \text{ and} \\ & \neg\exists j > i.\mathtt{R}_j\ \mathtt{rev\ C} \end{cases} \\
next(\mathtt{C}, \overline{\mathtt{R}}) & = & fstrev(\mathtt{D}, \overline{\mathtt{R}}) \quad (\text{if } \mathtt{C\ ext\ D})
\end{array}
$$

(Here, (i, j) stands for the set $\{i+1, \ldots, j-1\}$). We often omit the second argument $\overline{\mathtt{R}}$ to these functions when it is $dom(RT)$.

The function $mtype(\mathtt{m}, \mathtt{L}, \overline{\mathtt{R}}, \mathtt{L}')$ returns the signature $\overline{\mathtt{C}}\to\mathtt{C}$ of the method found in class L using $\overline{\mathtt{R}}$. $\mathtt{L}'$ represents the location of the caller. It is defined by the following rules:

$$
\frac{\mathtt{class\ L\ \{extends, revises\}\ C\ using}\ \overline{\mathtt{R}}'\ \mathtt{\{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}}\quad \mathtt{B\ m(}\overline{\mathtt{B}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}}\ [\mathtt{within\ L}'] \in \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{L}, \overline{\mathtt{R}}, \mathtt{L}') = \overline{\mathtt{B}}\to\mathtt{B}}
$$

$$
\frac{\mathtt{class\ L\ \{extends, revises\}\ C\ using}\ \overline{\mathtt{R}}'\ \mathtt{\{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}}\quad \mathtt{m}\ [\mathtt{within\ L}'] \notin \overline{\mathtt{M}}\quad mtype(\mathtt{m}, next(\mathtt{L}, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}') = \overline{\mathtt{B}}\to\mathtt{B}}{mtype(\mathtt{m}, \mathtt{L}, \overline{\mathtt{R}}, \mathtt{L}') = \overline{\mathtt{B}}\to\mathtt{B}}
$$

The first rule represents the case where m is found in L. The method may be a within method, in which case the location has to agree with the last argument. The second rule is for the case where m is not present in L ($\mathtt{m}\ [\mathtt{within\ L}'] \notin \overline{\mathtt{M}}$ means that there is neither method named m nor `m ··· within` $\mathtt{L}'$ in $\overline{\mathtt{M}}$); then, the signature is equivalent to that from the next class, represented by $next(\mathtt{L}, \overline{\mathtt{R}})$. As we see in typing rules, $\overline{\mathtt{R}}$ will be taken from the `using` clause of the class in which a method invocation appears so that typechecking of expressions does not need all revisers, that is, it is modular. As in FJ, we assume that `Object` has no methods and so $mtype(\mathtt{m}, \mathtt{Object}, \bullet, \mathtt{L})$ is undefined for any L.

Finally, we define the function $mbody(\mathtt{m}, \mathtt{L}, \mathtt{L}')$ to look up a method body. It returns the body of the method m in L called from $\mathtt{L}'$ as the triple, written $\overline{\mathtt{x}}\mathtt{.e\ in\ L}''$, where $\overline{\mathtt{x}}$ are parameters, e is the method body, and the location $\mathtt{L}''$ stands for the location where the method is found. The rules are similar to *mtype*:

$$
\frac{\mathtt{class\ L\ \{extends, revises\}\ C\ using}\ \overline{\mathtt{R}}\ \mathtt{\{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}}\quad \mathtt{B\ m(}\overline{\mathtt{B}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}}\ [\mathtt{within\ L}'] \in \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{L}, \mathtt{L}') = \overline{\mathtt{x}}\mathtt{.e\ in\ L}}
$$

$$
\frac{\mathtt{class\ L\ \{extends, revises\}\ C\ using}\ \overline{\mathtt{R}}\ \mathtt{\{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}}\quad \mathtt{m}\ [\mathtt{within\ L}'] \notin \overline{\mathtt{M}}\quad mbody(\mathtt{m}, next(\mathtt{L}), \mathtt{L}') = \overline{\mathtt{x}}\mathtt{.e\ in\ L}''}{mbody(\mathtt{m}, \mathtt{L}, \mathtt{L}') = \overline{\mathtt{x}}\mathtt{.e\ in\ L}''}
$$

Unlike *mtype*, however, it (implicitly) uses all revisers (remember that $next(\mathtt{L})$ is shorthand for $next(\mathtt{L}, dom(RT))$).

4.3 Type system

The subtype relation is written `C <: D`, which is the reflexive and transitive closure of the `extends` relation.

The type judgment for expressions is of the form $\mathtt{L}; \Gamma \vdash \mathtt{e : C}$, read "expression e in class/reviser L is given type C under type environment Γ." A type environment Γ, also written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{C}}$, is a finite mapping from variables $\overline{\mathtt{x}}$ to types $\overline{\mathtt{C}}$. The typing rules are given below.

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

$$
\frac{\mathtt{L}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad fields(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{L}; \Gamma \vdash \mathtt{e}_0\mathtt{.f}_i : \mathtt{C}_i} \qquad \text{(T-FIELD)}
$$

$$
\frac{\mathtt{L}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathtt{L}; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \mathtt{L\ using}\ \overline{\mathtt{R}} \quad mtype(\mathtt{m}, fstrev(\mathtt{C}_0, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{D}}\to\mathtt{C} \qquad \overline{\mathtt{C}}\ \mathtt{<:}\ \overline{\mathtt{D}}}{\mathtt{L}; \Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}} \qquad \text{(T-INVK)}
$$

$$
\frac{fields(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \mathtt{L}; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}}\ \mathtt{<:}\ \overline{\mathtt{D}}}{\mathtt{L}; \Gamma \vdash \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}} \qquad \text{(T-NEW)}
$$

$$
\frac{\mathtt{L}'; \Gamma \vdash \mathtt{e} : \mathtt{C}}{\mathtt{L}; \Gamma \vdash \mathtt{e\ in\ L}' : \mathtt{C}} \qquad \text{(T-IN)}
$$

Most rules are straightforward adaptations from FJ typing rules. In the rule T-INVK, method lookup starts from $\mathit{fstrev}(\mathtt{C}, \overline{\mathtt{R}})$, taking into account revised classes $\overline{\mathtt{R}}$ taken from the `using` clause of the current class `L`, which is also given to *mtype* as the caller information. In the rule T-IN, the location for `e` is switched since `e` originates from a method in `L′`.

The type judgment for methods is of the form `M OK IN L`, read "method `M` is well typed in class `L`." We show only the rule for methods in a reviser since the other rule for ones in a class is similar.

$$\frac{\begin{array}{c}\mathtt{R\ rev\ D} \qquad \mathtt{R}; \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{D} \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ \text{for any } \mathtt{L}, \text{ if } \mathit{mtype}(\mathtt{m}, \mathit{next}(\mathtt{R}), \mathit{dom}(RT), \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{D}_0, \\ \text{then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0\mathtt{;}\ \}\ [\mathtt{within\ L}']\ \mathtt{OK\ IN\ R}} \text{(T-METHODR)}$$

In both rules, the method body `e` has to be well typed under the type declarations of the parameters $\overline{\mathtt{x}}$; the type of `this` is the (revised, if the method is declared in a reviser) class name in which the method is declared. The last conditional premise checks whether `M` correctly overrides all the method (be it normal or within) of the same name in super classes. Note that $\mathit{dom}(RT)$ is used here. It means that it requires all revisers to check valid method overriding. Only this condition prevents completely modular typechecking and thus the corresponding check is deferred to the final stage of compilation (see the next section).

Finally, the type judgment for classes is written `CL OK`, meaning "class `CL` is well typed." We show only the rule for classes here (the other rule for revisers is similar).

$$\frac{\overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}}{\mathtt{class\ C\ extends\ D\ using}\ \overline{\mathtt{R}}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}}$$

A class table *CT* or *RT* is well typed if all the classes in it are well typed and we write (CT, RT) `OK` when both class tables are well typed.

4.4 Operational semantics

The reduction relation is of the form $\mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to expression `e′` in one step in `L`." Reduction rules are given below:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{L} \vdash \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{.f}_i \longrightarrow \mathtt{v}_i} \text{(R-FIELD)}$$

$$\frac{}{\mathtt{L} \vdash \mathtt{v\ in\ L}' \longrightarrow \mathtt{v}} \text{(R-RETURN)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathit{fstrev}(\mathtt{C}), \mathtt{L}) = \overline{\mathtt{x}}\mathtt{.e}_0\ \mathtt{in\ L}'}{\begin{array}{l}\mathtt{L} \vdash \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{.m}(\overline{\mathtt{w}}) \\ \longrightarrow (\lfloor \overline{\mathtt{w}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{v}})/\mathtt{this} \rfloor \mathtt{e}_0)\ \mathtt{in\ L}'\end{array}} \text{(R-INVK)}$$

All rules are straightforward. When a method is invoked on an object of `C`, method lookup starts from $\mathit{fstrev}(\mathtt{C})$, the first reviser for `C`. To distinguish the location of the method body from that of its caller, `in L′` is added to it. The rule R-RETURN represents the return from a method. Unlike FJ, the semantics is call-by-value because the location where an expression is reduced is important and passing a non-value expression from one location to another changes its meaning. So, the receiver and arguments must be a value, whose meaning is independent of locations, in R-FIELD and R-INVK.

We omit congruence rules, which allow a subexpression to reduce.

4.5 Type soundness

It is not difficult to show that GluonFJ is type sound; it enjoys subject reduction and progress [46], which can be proved in the standard manner. (See the companion technical report [10] for details.) Here, we suppose that (CT, RT) `OK` and write $\mathtt{L} \vdash \mathtt{e}_1 \longrightarrow^* \mathtt{e}_n$ when $\mathtt{L} \vdash \mathtt{e}_1 \longrightarrow \mathtt{e}_2, \ldots, \mathtt{L} \vdash \mathtt{e}_{n-1} \longrightarrow \mathtt{e}_n$.

THEOREM 1 (Subject Reduction). *If* $\mathtt{L}; \Gamma \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, *then* $\mathtt{L}; \Gamma \vdash \mathtt{e}' : \mathtt{D}$ *for some* `D` *such that* $\mathtt{C} <: \mathtt{D}$.

THEOREM 2 (Progress). *If* $\mathtt{L}; \emptyset \vdash \mathtt{e} \in \mathtt{C}$, *then either* `e` *is a value, or there exists some* `e′` *such that* $\mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$.

THEOREM 3 (Type Soundness). *If* $\mathtt{L}; \emptyset \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{L} \vdash \mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* `e′` *a normal form, then* `e′` *is either a value* `v` *with* $\mathtt{L}; \emptyset \vdash \mathtt{v} : \mathtt{D}$ *for some* $\mathtt{D} <: \mathtt{C}$.

PROOF 1. *Immediate from Theorems 1 and 2.*

5. Discussion

Mostly Modular typechecking

The typechecking of a GluonJ program is mostly modular. As we rigidly showed in the previous section, every class and reviser is modularly typecheckable except two rules. First, the precedence order among the revisers revising the same class must be a total order. Checking this rule requires all the revisers used in a program are known. Second, T-METHODR shown in Section 4.3 (and the corresponding rule for methods in a class) must be checked. This rule is violated when a method added by a reviser is incompatibly overridden by a method in an existing subclass (or another reviser for a subclass). Checking this also requires all the revisers are known at compile time.

The modularly typecheckable part of a program can be regarded as a module interface, which does not change by deployment configuration of revisers. On the other hand, the unmodularity of GluonJ is due to checking T-METHODR and the precedence order among revisers. This check guarantees compatibility among the modifications by revisers

since the modifications mutually affect each other. This check is a cost of dealing with crosscutting concerns.

However, the unmodularity of GluonJ is a property gradually introduced depending on language design. In fact, avoiding the unmodularity of GluonJ is possible although it restricts practical usefulness of the language. For example, a compiler could modularly typecheck T-MethodR if the language semantics were modified. We could avoid this unmodularity if method *overloading* considered return types as well as parameter types, for example, if we allow overloading two methods such as `int foo()` and `String foo()` when one is declared in a class and the other is in a subclass of that class. However, this new semantics is incompatible to normal Java's. Avoiding the global check of the precedence order is also possible if we restrict usefulness of GluonJ. For example, if a program can include only one reviser R and its directly/indirectly requiring revisers, a compiler can typecheck that the precedence order among those revisers are total order when it compiles the *root* reviser R.

Is encapsulation broken?

Since a reviser destructively modifies a class, like AspectJ, it might seem to break modularity. However, comparing GluonJ with other OOP languages, we can hardly claim that GluonJ largely breaks modularity. We see its AOP capability is provided by simple variants of OOP constructs like method overriding. It is just design decision to what extent the modularity is weakened from the rigid modularity achieved in OOP. Although GluonJ still has the fragile pointcut problem due to the `within` predicate and GluonJ's destructive nature may be dangerous, these problems are somewhat addressed by the normal visibility rule by `private` and so on. Typechecking is only mostly modular but, if we really need, we can fix this problem by giving up some expressive power of the language.

Our observation is that AOP is a paradigm for (not breaking but) loosing modularity to meet practical demands. The degree of loosing is a design issue. Since modularity is loosen with respect to typechecking and compilation, AOP requires more global knowledge for reasoning than OOP. However, OOP also requires global knowledge to a certain degree. To know the exact behavior of a method call in OOP, we need global knowledge, such as which class overrides the method.

6. Implementation

6.1 Compilation overview

We implement a reviser by translating it into a subclass of its target class and then adjusting a whole program so that the reviser will be used instead of its target class in instantiations. Thus, the compilation of a GluonJ program consists of two stages: source-to-bytecode translation and linking. We implemented the translation stage by extending JastAddJ [14] and the linking stage by using Javassist [8]. The linking stage is bytecode transformation and it is executed at load time or statically at the end of compile time. The source code of the compiler is available from our website.[2]

Translation stage

At the translation stage, a source file is separately compiled into Java bytecode. This compilation is modular; it needs only the revisers specified by `using` declarations and `requires` clauses as well as the classes and interfaces on which the source file explicitly depends. Static typechecking is executed at this stage except T-METHODR shown in Section 4.3. Informally, our compiler performs the typechecking for normal Java and generates bytecode as if a reviser is a direct subclass of its target class. It also translates a member-access expression if it accesses a member added by a reviser. For example, an expression `e.print()` is translated into:

```
((Printing)e).print()
```

if the `print` method is newly added to the class of `e` by the reviser `Printing` given by a `using` declaration (as in Figure 8). A `within` method is compiled as if it is a normal method. The `within` predicate is translated into a Java annotation to that method.

A constructor of a reviser is compiled differently from normal constructors of Java classes. Recall that a reviser cannot declare an explicit constructor. At this stage, the compiler generates only the default constructor, which takes no arguments, initializes field values, and calls the default constructor of the target class since the target class of a reviser is treated as its super class during compilation. The target class does not have to declare the default constructor since the generated constructor is modified at the next stage.

Linking stage

At the linking stage, all the revisers included in a program must be given. In this sense, this stage needs a whole-program analysis and hence it is not modular. Our linker applies bytecode transformation to every compiled class. Since this transformation can be applied separately to every class if only all the revisers are given, this stage is still fairly modular. It never refers to other classes.

Our linker first computes the precedence order among the revisers and checks its validity. This may throw a compile error. Then the linker sets the direct super class of every reviser to its target class. If multiple revisers revise the same target class, then they are linearized to make a single inheritance hierarchy according to the precedence order. For example, if class R and S revises the same class C and the reviser R has higher precedence than S, then R extends S, which extends C. If the target class has normal subclasses, their super class is changed from that target class to the reviser. If a normal class D extends C, then D is also changed to extend R. When a super class is changed, all method calls on `super` is

[2] www.csg.is.titech.ac.jp/projects/gluonj

also modified (at bytecode level) to invoke a method in the new super class.

Our linker next checks the last premise of T-METHODR, which requires that the method overriding by revisers is valid. Suppose that two revisers revise the same class. If these revisers declare methods with the same name and parameter types but with different (not-covariant) return types, then this method overriding is invalid and a compile error will be thrown.

The linker also generates constructors for every reviser by modifying the default constructor generated at the first stage. A constructor is generated per constructor of the super class. It calls super() with the received arguments and then initializes field values. For example, the reviser FloatExt in Figure 5 will have the following constructor if its target class AddExpr has a constructor `AddExpr(ASTree,ASTree)`:

```
public FloatExt(ASTree left, ASTree right) {
  super(left, right);
  // initialize field values
}
```

The generated constructors are public. The constructors of the target class may be changed to public or protected to be visible from the reviser.

Finally, our linker applies bytecode transformation to every class file. If a class C is revised by a reviser R, then all instantiations of C is transformed into instantiations of R. If C is revised by multiple classes, then they are instantiations of the reviser with the highest precedence. Furthermore, if a static method is overridden by a reviser, then all occurrences of the call to that static method is redirected to the overriding method in the reviser.

Implementing a within method

If there is a within method, method dispatch considers a triple: method name, receiver type, and caller location. In our implementation, the method name and the caller location are statically evaluated and only the receiver type is dynamically evaluated at every method call. The overhead of calling a within method is, therefore, equivalent to that of calling a normal method, in which only the receiver type must be dynamically evaluated. Although caller location is static, this optimization is not naive since a within method may be overridden by a normal method in a subclass.

Suppose that a method foo declared in R has a predicate within L. Our linker first renames that within method to foo_L (the real method name is more elaborate). Next, our linker transforms a call to foo into a call to foo_L if that call expression is located within L and the static type of the receiver is R or a super type of R. Otherwise, the call is not transformed. Since the dynamic type of that receiver may not be R, if a super type or a subtype of R declares a method foo, then our linker generates a method foo_L for that type. Here, a super/sub type of R is determined on the basis of the inheritance relations modified at the linking stage. A super type may be an interface. The generated foo_L method just executes the foo method declared in that same class by the invokespecial bytecode (or the body of foo_L is a copy of the body of foo). This delegation may cause a small runtime overhead. Furthermore, if a super type of R declares a foo method, then all the subtypes of that type are transformed to have a foo_L method. If the super type is an interface, its subtypes include classes implementing that interface.

Since Java prohibits transforming some built-in classes, our implementation does not allow revising a class if it is instantiated within those built-in classes. The linker cannot modify this instantiation. It allows declaring a within method that overrides a method in a built-in class, such as the toString method in the Object class, but that within method is implemented in a slightly different approach. Suppose that a reviser FloatExt revises the AddExpr class and it declares the toString method with a predicate within Parser. According to the approach mentioned above, our linker must append a toString_Parser method to the Object class. However, since this is not possible, our linker transforms a call to toString within the Parser class in a different approach. For example, if the expression is `expr.toString()` and the type of expr is a super type of AddExpr, then the linker transforms it into this:

```
expr instanceof AddExpr
  ? ((AddExpr)expr).toString_Parser()
  : expr.toString()
```

This does not call the toString_Parser method if the receiver does not understand that method. The condition expression will be more complex when a sibling of AddExpr is revised to have a toString method with the same predicate.

6.2 Formal model of compilation

We formalize the core of the implementation scheme described above as translation from GluonFJ to FJ. Instead of reviewing the definition of FJ from scratch, we use a subset of GluonFJ, where the reviser class table is empty, as the target language. (Precisely speaking, we need typecasts `(C)e` in the target language but omit typing and reduction rules since they can be added in a straightforward manner as in FJ [24].) For simplicity, the formalized translation is done at once, mixing the two stages described above. After giving the definition of formal translation, we state that translation preserves typing.

First, we give a few auxiliary definitions used in the translation. The function $origin(\mathtt{m}, \mathtt{L})$ returns the name of the super class of L that has first introduced m (be it normal or within) in the class hierarchy.

$$\frac{\texttt{class L } \{\texttt{extends}, \texttt{revises}\} \texttt{ D using } \overline{\texttt{R}}' \texttt{ } \{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\} \quad \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\texttt{\{ return e; \}}\ [\texttt{within } \texttt{L}_1] \in \overline{\texttt{M}} \quad \text{for any } \texttt{L}_0, mtype(\mathtt{m}, next(\mathtt{L}), dom(RT), \mathtt{L}_0) \text{ undefined}}{origin(\mathtt{m}, \mathtt{L}) = \mathtt{L}}$$

$$\frac{\mathit{origin}(\mathtt{m}, \mathit{next}(\mathtt{L})) = \mathtt{L}'}{\mathit{origin}(\mathtt{m}, \mathtt{L}) = \mathtt{L}'}$$

The next two functions $\mathit{within}_{sub}(\mathtt{m}, \mathtt{C})$ and $\mathit{within}(\mathtt{m}, \mathtt{L})$ are used to translate within methods. The function $\mathit{within}_{sub}(\mathtt{m}, \mathtt{C})$ collects all predicates L from classes that revise a subclass of C and contain a within method of name m.

$$\mathit{within}_{sub}(\mathtt{m}, \mathtt{C}) = \left\{ \mathtt{L} \;\middle|\; \begin{array}{l} \mathtt{R} \in \mathit{dom}(RT) \text{ and } (\exists \mathtt{D}.\mathtt{R}\ \mathtt{rev}\ \mathtt{D}\ \mathtt{<:}\ \mathtt{C}) \\ \text{and } \mathtt{class\ R\ revises\ \cdots\ \{\ \overline{M}\ \}} \\ \text{and } \mathtt{B\ m(\overline{B}\ \overline{x})\{return\ e;\}\ within\ L} \in \overline{\mathtt{M}} \end{array} \right\}$$

The function $\mathit{within}(\mathtt{m}, \mathtt{L})$, which collects all locations used as predicates for m in L, is defined to be $\mathit{within}_{sub}(\mathtt{m}, \mathit{thistype}(\mathit{origin}(\mathtt{m}, \mathtt{L})))$ where $\mathit{thistype}(\mathtt{C}) = \mathtt{C}$ and $\mathit{thistype}(\mathtt{R}) = \mathtt{C}$ if R rev C. For example, consider the following classes:

```
class A extends Object { C m(){ return e1; } }
class B extends A { C m(){ return e2; }}
class R1 revises A {
  C m(){ return e3; } within C
}
class R2 revises B {
  C m(){ return e4; } within D
}
```

Then, $\mathit{within}_{sub}(\mathtt{m}, \mathtt{B}) = \{\mathtt{D}\}$ and $\mathit{within}(\mathtt{m}, \mathtt{B}) = \mathit{within}(\mathtt{m}, \mathtt{R2}) = \mathit{within}(\mathtt{m}, \mathtt{R1}) = \mathit{within}(\mathtt{m}, \mathtt{A}) = \{\mathtt{C}, \mathtt{D}\}$. When $\mathtt{L} \in \mathit{within}(\mathtt{m}, \mathtt{C})$ the invocation of m on C from class L will be translated to the invocation of $\mathtt{m}_\mathtt{L}$, which stands for the name mangled from m and L.

The judgment for translation is of the form $\mathtt{L}; \Gamma \vdash \mathtt{e} \Longrightarrow \mathtt{e}'$, read "expression e is translated to $\mathtt{e}'$ under L and Γ." We show only rules for method invocations and constructors, since others are trivial. In translating a method invocation, we ensure that a method only available in a reviser can be invoked, by casting the receiver to $\mathtt{L}'$, which is the reviser that revises the receiver type with the highest priority. As we mentioned above, the method name may be changed if the caller location L is found in $\mathit{within}(\mathtt{m}, \mathtt{C}_0)$.

$$\frac{\begin{array}{c}\mathtt{L}; \Gamma \vdash \mathtt{e}_0 \Longrightarrow \mathtt{e}_0' \qquad \mathtt{L}; \Gamma \vdash \overline{\mathtt{e}} \Longrightarrow \overline{\mathtt{e}}' \qquad \mathtt{L}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \\ \mathtt{L\ using\ \overline{R}} \qquad \mathit{fstrev}(\mathtt{C}_0, \overline{\mathtt{R}}) = \mathtt{L}' \\ \mathtt{m}' = \begin{cases} \mathtt{m}_\mathtt{L} & \text{if } \mathtt{L} \in \mathit{within}(\mathtt{m}, \mathtt{C}_0) \\ \mathtt{m} & \text{otherwise} \end{cases}\end{array}}{\mathtt{L}; \Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \Longrightarrow ((\mathtt{L}')\mathtt{e}_0').\mathtt{m}'(\overline{\mathtt{e}}')} \text{(TR-INVK)}$$

For example, under the four classes A, B, R1, and R2 above, $\mathtt{D}; \mathtt{x}\mathtt{:}\ \mathtt{B} \vdash \mathtt{x.m()} \Longrightarrow \mathtt{((R2)x).m_D()}$ is derivable. The translation of object creation is mostly trivial, except that the class name is changed to $\mathit{fstrev}(\mathtt{C})$.

$$\frac{\mathit{fstrev}(\mathtt{C}) = \mathtt{L}' \qquad \mathtt{L}; \Gamma \vdash \overline{\mathtt{e}} \Longrightarrow \overline{\mathtt{e}}'}{\mathtt{L}; \Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \Longrightarrow \mathtt{new\ L}'(\overline{\mathtt{e}}')}$$

The judgment for translation of methods is written $\mathtt{L} \vdash \mathtt{M} \Longrightarrow \overline{\mathtt{M}}$. Note that the translation of one method may result in multiple methods. The first rule is for translation of a normal method. The method body is translated under the type environment where parameters have declared types and this has $\mathit{thistype}(\mathtt{L})$, which is equivalent to the static type used in typechecking. Since a normal method is called "within everywhere," it overrides all the other within methods, which have mangled names. The location names are collected by $\mathit{within}(\mathtt{m}, \mathtt{L})$.

$$\frac{\mathtt{L}; \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{B}}, \mathtt{this}\mathtt{:}\mathit{thistype}(\mathtt{L}) \vdash \mathtt{e} \Longrightarrow \mathtt{e}' \qquad \mathit{within}(\mathtt{m}, \mathtt{L}) = \overline{\mathtt{L}}}{\begin{array}{rl} & \mathtt{L} \vdash \mathtt{B\ m(\overline{B}\ \overline{x})\{\ return\ e;\ \}} \\ & \quad \mathtt{B\ m(\overline{B}\ \overline{x})\{\ return\ e';\ \}} \\ & \quad \mathtt{B\ m_{L_1}(\overline{B}\ \overline{x})\{\ return\ e';\ \}} \\ \Longrightarrow & \quad \vdots \\ & \quad \mathtt{B\ m_{L_n}(\overline{B}\ \overline{x})\{\ return\ e';\ \}} \end{array}}$$

For example, the method m in class B above will translate to three methods named m, $\mathtt{m}_\mathtt{D}$, and $\mathtt{m}_\mathtt{C}$, even though no reviser revising B has a method with within C. (B will have R1, which has $\mathtt{m}_\mathtt{C}$ after translation, as a super class of B.) The translation of a within method is straightforward.

$$\frac{\mathtt{L}; \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{B}}, \mathtt{this}\mathtt{:}\mathit{thistype}(\mathtt{L}) \vdash \mathtt{e} \Longrightarrow \mathtt{e}'}{\begin{array}{c}\mathtt{L} \vdash \mathtt{B\ m(\overline{B}\ \overline{x})\{\ return\ e;\ \}\ within\ L'} \\ \Longrightarrow \mathtt{B\ m_{L'}(\overline{B}\ \overline{x})\{\ return\ e';\ \}}\end{array}}$$

Translation of a class is written $\vdash \mathtt{CL} \Longrightarrow \mathtt{CL}'$, and the translation rule is below.

$$\frac{\mathit{next}(\mathtt{L}) = \mathtt{L}' \qquad \mathtt{L} \vdash \overline{\mathtt{M}} \Longrightarrow \overline{\mathtt{M}}'}{\begin{array}{c}\vdash \mathtt{class\ L\ \{extends, revises\}\ D\ using\ \overline{R}\ \{\ \overline{C}\ \overline{f};\ \overline{M}\ \}} \\ \Longrightarrow \mathtt{class\ L\ extends\ L'\ \{\ \overline{C}\ \overline{f};\ \overline{M}'\ \}}\end{array}}$$

The super class D is replaced with $\mathtt{L}'$, which is the name of the next class when looking up definitions. Note that every reviser is translated to a normal class, so, precisely speaking, the set of class names of the target language is taken as the union of the sets of class and reviser names of the source language.

We write $(CT, RT) \Longrightarrow CT'$ when every class in the source class tables translates to one in the target, namely, (1) $\mathit{dom}(CT') = \mathit{dom}(CT) \cup \mathit{dom}(RT)$; (2) $\vdash CT(\mathtt{C}) \Longrightarrow CT'(\mathtt{C})$ for any $\mathtt{C} \in \mathit{dom}(CT)$; and (3) $\vdash RT(\mathtt{R}) \Longrightarrow CT'(\mathtt{R})$ for any $\mathtt{R} \in \mathit{dom}(RT)$.

6.3 Properties of Translation

We show that translation preserves typing. Actually, translation does not change the type of an expression, except for the case where the expression is $\mathtt{new\ C}(\overline{\mathtt{e}})$, in which case it will become $\mathit{fstrev}(\mathtt{C})$. Since we mention two programs before and after translation at the same time, we explicitly say

```
class C {
  public void work(T1 ticker) {
    ticker.tick();
  }
}

class T1 {
  public void tick() {
    Microbench.count1 += 1;
  }
}

class R revises T1 {
  public void tick() within C {
    Microbench.count2 += 1;
  }
}

class T2 extends T1 {
  public void tick() {
    Microbench.count1 += 1;
  }
}
```

Figure 9. Microbenchmark workload

which class table is assumed to avoid confusion. Since locations are not significant in the target program, we omit L from judgments.

LEMMA 1. *If* $(CT, RT) \Longrightarrow CT'$ *and* $\mathtt{L};\Gamma \vdash \mathtt{e} : \mathtt{C}$ *under* (CT, RT) *and* $\mathtt{L};\Gamma \vdash \mathtt{e} \Longrightarrow \mathtt{e}'$, *then* $\Gamma \vdash \mathtt{e}' : \mathtt{L}$ *(under* CT'*) where* L *is either* C *or fstrev*(C).

It is easy to show that translation succeeds when both class and reviser tables are well typed. Then, we now have the theorem that a pair of well-typed class and reviser tables translates to a well-typed class table.

THEOREM 4. *If* (CT, RT) *is OK, then there exists* CT' *such that* $(CT, RT) \Longrightarrow CT'$ *and* CT' *is OK.*

It is left for future work to prove that the translation also preserves semantics, meaning that no typecasts inserted by the translation will fail.

6.4 Experiments

As we showed above, since within methods are transformed into normal method calls, the execution overheads due to them are extremely small. To investigate this fact, we performed series of experiments on a machine with Intel Core 2 Duo E8500 3.16GHz processor, 3GB memory, Gentoo Linux with libc ver. 2.9, and Java 1.6.0_15 HotSpot Server VM (build 14.1-b02, mixed mode).

Microbenchmark

We evaluated the technique adopted by GluonJ as well as several other possible implementation techniques. For comparison, we first implemented a micro benchmark incorporating several implementation techniques for within methods. It uses two classes T1 and T2 and one reviser R (see Figure 9). T2 extends T1 and R revises T1. Each

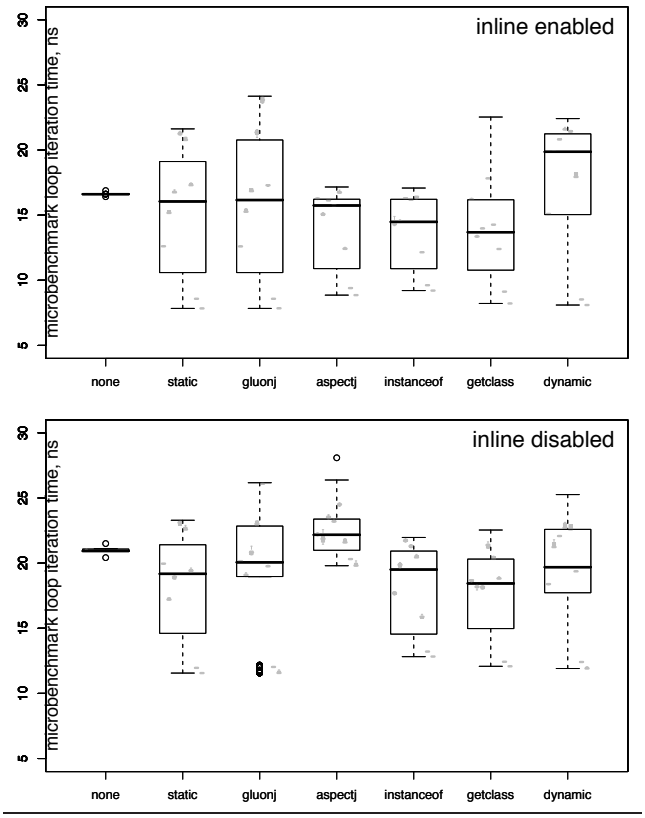

Figure 10. Overheads of method dispatch techniques.

class/reviser declares a method tick and R's method has a predicate within C. The benchmark calls tick from C on a various mixture of objects with static type T1. This operation is repeated 10 million times in a hot loop, compiled by a JIT (Just-In-Time) compiler with the highest optimization level. We found in initial experiments that uniform or close to uniform distribution of object types results in a JIT compiler using additional optimizations and hence the comparison becomes biased towards cases with less object diversity. To reduce this bias, callee objects of different types are pseudo-randomly chosen at every call to tick.

Figure 10 shows the results. The upper graph shows the results when the code inlining by a JIT compiler is enabled. The lower graph shows the results when it is disabled. The vertical axis represents the duration of a single iteration of micro benchmark in nano seconds. The horizontal axis represents implementation techniques for within methods. We examined seven techniques: *none*, *gluonj*, *static*, *aspectj*, *instanceof*, *getclass*, and *dynamic*. Gray dots visible on the graphs show the data points of measurement with a particular ratio of object mixture.

When optimizations of the caller site are possible, like guarded devirtualization, all implementation techniques exhibit similar performance. *none* presents a reference point, where the method is not overridden. *gluonj* presents our implementation. The reviser R is translated into a subclass of

T1 (and a super class of T2). A within method tick is renamed into tick_C and a call by C to tick is translated into a call to tick_C. Thus, a tick_C method is also added to T1 and T2. *static* is the same as gluonj except the body of a tick_C method in T1 and T2 is a copy of tick and thus it implies no overhead due to delegation. *aspectj* presents an equivalent implementation in AspectJ. In *instanceof* and *getclass*, the caller calls tick_C only when the receiver object is R; otherwise, it calls tick. *instanceof* checks the type by the instanceof operator:

```
public void work(T1 ticker) {
  if (ticker instanceof R
      && !(ticker instanceof T2))
    ticker.tick_C();
  else
    ticker.tick();
}
```

On the other hand, *getclass* does this by the getClass method:

```
public void work(T1 ticker) {
  if (ticker.getClass() == R.class)
    ticker.tick_C();
  else
    ticker.tick();
}
```

Finally, in *dynamic*, the caller object passes a Class object representing the type of the caller and the within method tick in R checks it:

```
public void tick(Class caller) {
  if (caller == C.class)
    Microbench.count2 += 1;
  else
    Microbench.count1 += 1;
}
```

The DaCapo benchmarks

Next, we used the DaCapo suite of benchmarks [5] to evaluate potential impact of within methods on real applications, especially in the cold code. To conservatively estimate the impact, we directly instrumented the bytecode of all application methods, which is structurally equivalent to an "empty" within method. Thus, all method calls first invoke that within method, which delegates to the original one. We compared several of the implementation techniques mentioned above, those that can be easily imitated via code instrumentation.

Figure 11 shows the results on 8 of 11 DaCapo benchmarks (the other benchmarks did not work after the instrumentation). The vertical axis represents the elapsed time of each run in seconds. A lower number is better. The horizontal axis represents the iteration of the run from the 1st to the 10th. A first few iterations show the performance before deep optimization by a JIT compiler.

Overhead of the instrumentation on the first iteration is up to 22% for well-behaving benchmarks, and up to 151% for `luindex`. Note that we measured overheads of extreme cases, in which all methods are within methods. In real applications, most methods would not be within ones. Since the modification to the benchmark bytecode is essentially redundant code, the impact is contained only in a warm-up phase except for `lusearch` and `luindex`, which have the same underlying software package, Lucene. The cause of such a drastic and constant slowdown remains a topic for further inquiry. However, in the two benchmarks, the implementation technique of GluonJ shows the best performance. For the other benchmarks, the impact of any implementation technique is negligible on the 10th iteration.

7. Related work

Predicate dispatch

The original predicate dispatch allows only predicates that access variables locally visible in the static scope of a method, such as parameters, the receiver object (*i.e.* this variable in Java), and their fields. For example, in JPred [32], a method can be selected only when a parameter value is an instance of some class. The predicates of JPred were carefully selected by the JPred designer for modular compilation.

On the other hand, GluonJ provides *contextual* predicate dispatch. The within predicate of GluonJ refers to external calling contexts, like *who is a caller*, since it must deal with crosscutting structures, which intrinsically depend on the externals. This fact removes some locality from methods and complicates modular compilation. However, as we showed above, the compilation of a GluonJ program by our compiler is still mostly modular.

Presenting similarity between predicate dispatch and the pointcut-advice of AOP is not new. This idea has been pointed out by other researchers [6, 21, 35] and ourselves [9]. The idea of method dispatch depending on a caller object is also found in [41]. Our contribution is that we actually design and implement a language based on this idea and discussed modular typechecking and compilation.

Other related work

A unique feature of GluonJ is to provide OOP-based mechanisms for not only enabling destructive extension but also limiting its scope. Although limiting its scope has not got much attention except in the AOP contexts, enabling destructive extension has been actively studied in OOP. For example, MixJuice [23], Feature-Oriented Programming (FOP) [3], and JavaGI [45] enable this as much as AOP languages. The partial classes of C# allow adding new members to an existing class. Categories in Objective-C also do so. Some researchers of FOP have suggested using AOP constructs such as pointcut and advice for crosscutting structures while using their OOP-based construct, which is a destructive style of mixin, for destructive extension [2]. Their proposal is a hybrid of OOP and AOP.

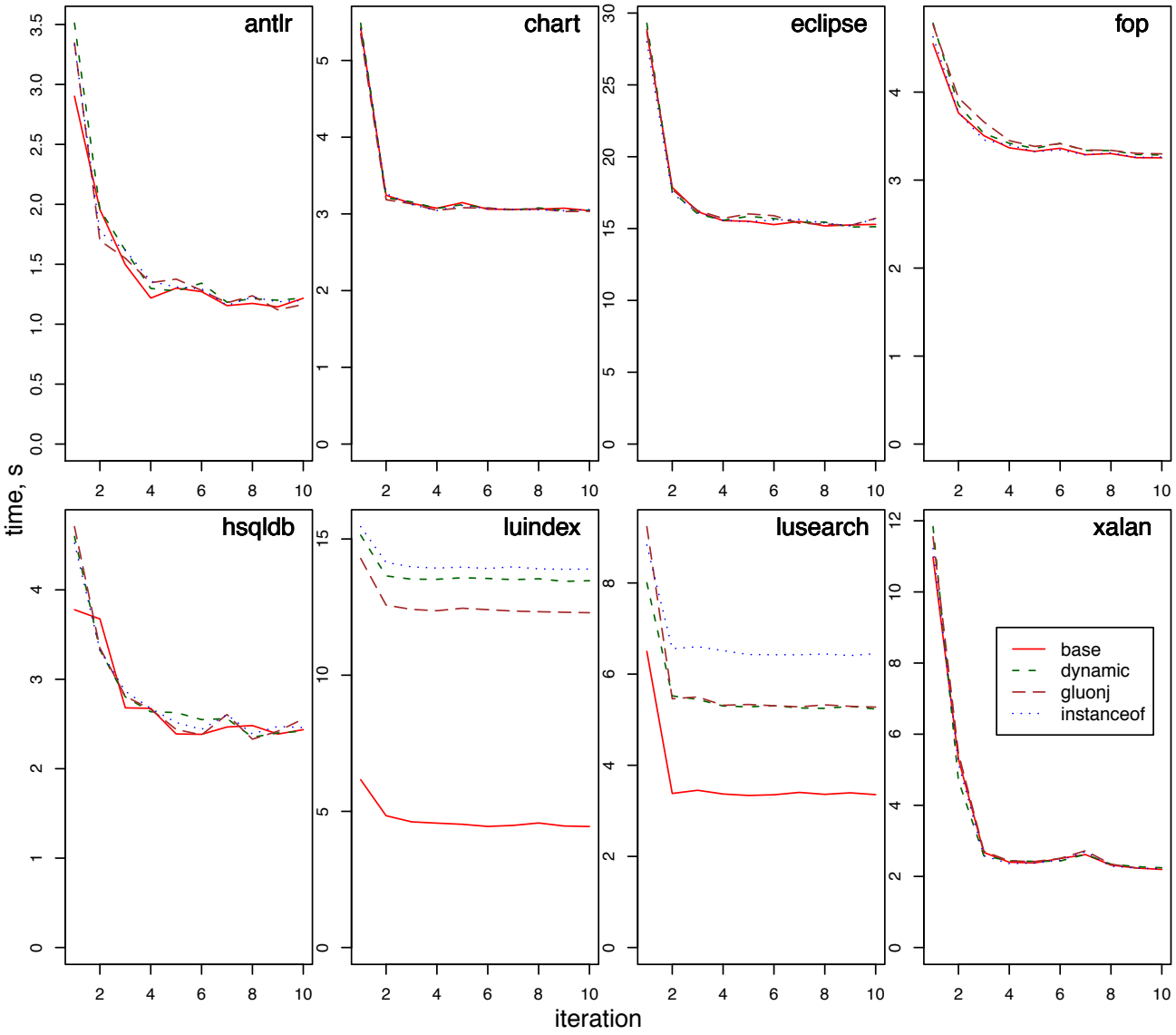

Figure 11. DaCapo benchmarks.

Open classes [12] and expanders [44] also enable destructive extension although they can only append new methods but cannot override existing methods. They are using declarations like using of GluonJ for modular typechecking. The work by Malabarba et al. [30] allows dynamically changing class definitions.

Classbox/J [4] and Context-Oriented Programming (COP) [22] can apply destructive extension into only a limited scope of program. However, their ability is not equivalent to the within/withincode pointcut of AspectJ or GluonJ's contextual predicate dispatch. They do not enable executing extra code in the middle of an existing method body without modifying the body. For example, in Classbox/J, a new extension is applied and made effective only within a package explicitly importing that extension. This import is described in the source code of the package, that is, the client-side code. On the other hand, in GluonJ, where a new extension is applied is described in the source code of that extension.

J&$_s$ [38] also supports destructive extension through the mechanism called class sharing. Although there is no need to modify the original program that creates an object of a modified class, a view change operation has to be applied to enable access to new members. From a typechecking point of view, the view change operation plays a role somewhat similar to using declarations, but the execution model is rather different since new members do not always override old ones.

AspectJ2EE [13] is an AOP system but it has similarity to GluonJ since it implements an aspect by a subclass of the target class. However, it only supports the execution

pointcut but not call or within pointcuts. Hence it cannot limit the scope of extension. It does not provide the same expressiveness that GluonJ does.

An AOP language *Hyper/J* [36] might seem to adopt OOP-style constructs as GluonJ does. It allows implementing every concern by a normal Java class like the partial classes of C#. However, how to *weave* (combine) multiple classes to compose a complete class must be described separately in a dedicated language, which is far from typical OOP languages like Java.

We have been developing a series of AOP languages and some of the languages inherited the name GluonJ. However, there is no overlap among this paper and the others. The design of those languages are different. The first GluonJ published in [11] used XML for describing aspects. The aim of this work was to allow programmers to flexibly control the construction of aspect instances. The second GluonJ published in [33] is more similar to GluonJ presented in this paper but it is a dynamic AOP language. The work focused on how to dynamically deploy intertype declarations.

8. Concluding remark

We presented GluonJ, which supports revisers and within methods for AOP. These mechanisms are natural enhancement to OOP ones and hence GluonJ can be compared with OOP languages on a side-by-side basis. We developed a formal system for GluonJ and rigidly presented that mostly modular typechecking and compilation is possible. Since the AOP capability of GluonJ is a subset of AspectJ's, extending it and discussing its modularity is our future work.

Through this work, we observed that AOP does not largely break modularity, relatively to OOP, but AOP looses it to meet demands. Since crosscutting modules may conflict each other, the compatibility among the modules must be checked in an unmodular fashion with global knowledge. This is a source of the unmodularity of AOP. It might be possible to avoid unmodular checking by restricting the expressive power of the language so that crosscutting modules will not conflict. Although this approach makes the language more modular, we would have to give up the flexibility for implementing destructive extension.

Our discussion in this paper is on modular typechecking and compilation; it is not on modular reasoning. However, our work would be a solid basis of further discussion of the modularity of AOP.

Acknowledgement

We thank Hidehiko Masuhara and other members of the Kumiki project for their helpful comments on this work. This work was partly sponsored by the KAKENHI program of JSPS/MEXT of Japan.

References

[1] Aldrich, J.: Open modules: Modular reasoning about advice. In: ECOOP 2005. pp. 144–168. LNCS 3586, Springer-Verlag (2005)

[2] Apel, S., Batory, D.: When to use features and aspects?: A case study. In: Proc. of the 5th Int'l Conf. on Generative Programming and Component Engineering (GPCE '06). pp. 59–68. ACM Press (2006)

[3] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling stepwise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2004)

[4] Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the scope of change in Java. In: Proc. of ACM OOPSLA. pp. 177–189 (2005)

[5] Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proc. of ACM OOPSLA. pp. 169–190. ACM (2006)

[6] Bockisch, C., Haupt, M., Mezini, M.: Dynamic virtual join point dispatch. Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '06) (2006)

[7] Bracha, G., Cook, W.: Mixin-based inheritance. In: Proc. of OOPSLA/ECOOP '90. pp. 303–311. ACM Press (1990)

[8] Chiba, S.: Load-time structural reflection in Java. In: ECOOP 2000. pp. 313–336. LNCS 1850, Springer-Verlag (2000)

[9] Chiba, S.: Predicate dispatch for aspect-oriented programming. In: the 2nd Workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms (VMIL '08). pp. 1–5. ACM (2008)

[10] Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular composition of crosscutting structures by contextual predicate dispatch. Research Reports C-267, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology (December 2009)

[11] Chiba, S., Ishikawa, R.: Aspect-oriented programming beyond dependency injection. In: ECOOP 2005. pp. 121–143. LNCS 3586, Springer-Verlag (2005)

[12] Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for Java. In: Proc. of ACM OOPSLA. pp. 130–145. ACM Press (2000)

[13] Cohen, T., Gil, J.Y.: AspectJ2EE = AOP + J2EE : Towards an aspect based, programmable and extensible middleware framework. In: ECOOP 2004 — Object-Oriented Programming. pp. 219–243. LNCS 3086 (2004)

[14] Ekman, T., Hedin, G.: The Jastadd extensible Java compiler. In: Proc. of ACM OOPSLA. pp. 1–18. ACM (2007)

[15] Ernst, E.: Family polymorphism. In: ECOOP 2001 — Object-Oriented Programming. pp. 303–326. LNCS 2072, Springer-Verlag (2001)

[16] Ernst, M., Kaplan, C., Chambers, C.: Predicate dispatching: A unified theory of dispatch. In: ECOOP '98 - Object-Oriented Programming. pp. 186–211. Springer-Verlag (1998)

[17] Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Filman, R.E., Elrad,

T., Clarke, S., Akşit, M. (eds.) Aspect-Oriented Software Development, pp. 21–35. Addison-Wesley (2005)

[18] Fraine, B.D., Südholt, M., Jonckers, V.: Strongaspectj: flexible and safe pointcut/advice bindings. In: Proc. of 7th Int'l Conf. on Aspect-Oriented Software Development (AOSD 2008). pp. 60–71. ACM (2008)

[19] Griswold, W.G., et al.: Modular software design with crosscutting interfaces. IEEE Software 23(1), 51–60 (2006)

[20] Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. In: Proc. of ACM OOPSLA. pp. 161–173 (2002)

[21] Haupt, M., Schippers, H.: A machine model for aspect-oriented programming. In: ECOOP 2007 – Object-Oriented Programming. LNCS, vol. 4609, pp. 501–524 (2007)

[22] Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (2008)

[23] Ichisugi, Y., Tanaka, A.: Difference-based modules: A class-independent module mechanism. In: ECOOP 2002 – Object-Oriented Programming. pp. 62–88. LNCS 2374 (2002)

[24] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Trans. Prog. Lang. Syst. 23(3), 396–450 (May 2001)

[25] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP'97 – Object-Oriented Programming. pp. 220–242. LNCS 1241, Springer (1997)

[26] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001 – Object-Oriented Programming. pp. 327–353. LNCS 2072, Springer (2001)

[27] Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proc. of the Int'l Conf. on Software Engineering (ICSE'05). pp. 49–58. ACM Press (2005)

[28] Koppen, C., Stoerzer, M.: Pcdiff: Attacking the fragile pointcut problem. In: Proc. of European Interactive Workshop on Aspects in Software (EIWAS'04) (2004)

[29] Lesiecki, N.: Improve modularity with aspect-oriented programming. `http://www.ibm.com/developerworks/java/library/j-aspectj` (2002)

[30] Malabarba, S., et al.: Runtime support for type-safe dynamic Java classes. In: ECOOP 2000. pp. 337–361. LNCS 1850, Springer-Verlag (2000)

[31] McEachen, N., Alexander, R.T.: Distributing classes with woven concerns: an exploration of potential fault scenarios. In: Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'05). pp. 192–200. ACM Press (2005)

[32] Millstein, T.: Practical predicate dispatch. In: Proc. of ACM OOPSLA. pp. 345–364. ACM (2004)

[33] Nishizawa, M., Chiba, S.: A small extension to Java for class refinement. In: Proc. of the 23rd ACM Sympo. on Applied Computing (SAC'08). pp. 160–165 (2008)

[34] Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: Proc. of ACM OOPSLA. pp. 99–115 (2004)

[35] Orleans, D.: Separating behavioral concerns with predicate dispatch, or, if statement considered harmful. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA '01 (2001)

[36] Ossher, H., Tarr, P.: Hyper/J: multi-dimensional separation of concerns for Java. In: Proc. of the Int'l Conf. on Software Engineering (ICSE). pp. 734–737 (2000)

[37] Parnas, D.L.: Information distributions aspects of design methodology. In: Proc. of IFIP Congress '71. pp. 26–30 (1971)

[38] Qi, X., Myers, A.C.: Sharing classes between families. In: Proc. of Conf. on Programming Language Design and Implementation. pp. 281–292 (2009)

[39] Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'03). LNCS, vol. 2743, pp. 248–274. Springer Verlag (July 2003)

[40] Smaragdakis, Y., Batory, D.: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol. 11(2), 215–255 (2002)

[41] Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. Theory and Practice of Object Systems 2(3), 161–178 (1996)

[42] Steimann, F.: The paradoxical success of aspect-oriented programming. ACM SIGPLAN Notices 41(10), 481–497 (2006)

[43] Stoerzer, M., Graf, J.: Using pointcut delta analysis to support evolution of aspect-oriented software. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance. pp. 653–656. IEEE Computer Society (2005)

[44] Warth, A., Stanojević, M., Millstein, T.: Statically scoped object adaptation with expanders. In: Proc. of ACM OOPSLA. pp. 37–56 (2006)

[45] Wehr, S., Lämmel, R., Thiemann, P.: JavaGI: Generalized interfaces for Java. In: ECOOP 2007 — Object-Oriented Programming. LNCS 4609, Springer-Verlag (2007), 347–372

[46] Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (Nov 1994)