ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

Mostly modular composition of crosscutting structures by contextual predicate dispatch (extended version)

Shigeru Chiba and Atsushi Igarashi and Salikh Zakirov

December 2009, C–267

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

Mostly modular composition of crosscutting structures by contextual predicate dispatch (extended version)

Shigeru Chiba[1], Atsushi Igarashi[2], and Salikh Zakirov[1]

[1] Grad. School of Info. Sciences and Engineering, Tokyo Institute of Technology
[2] Grad. School of Informatics, Kyoto University

Research Reports C-267
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology

Abstract. We propose an enhancement of method dispatch for enabling to compose both normal and crosscutting program structures. Our idea is to use predicate dispatch modified to refer to external calling contexts. Despite the support of crosscutting structures, our language based on this idea, named *GluonJ*, allows mostly modular typechecking and compilation. Its execution overhead is negligible. We show these facts through practice and theory.

1 Introduction

Aspect-oriented programming (AOP) has been actively studied for more than a decade. A challenge of AOP is to manage crosscutting structures in a modular way. Typical AOP languages such as AspectJ [27] have been tackling this by providing dedicated language constructs while they provide object-oriented programming (OOP) constructs like classes, methods, and inheritance. The most well-known language construct for AOP is pointcut-advice of AspectJ.

In this paper, we propose a natural enhancement of OOP language constructs to be able to manage crosscutting structures. We do not orthogonally introduce AOP-dedicated constructs into an OOP language. We instead enhance method dispatch to introduce AOP flavor into OOP. In our previous workshop paper [10], we presented that the language constructs of AspectJ can be emulated by predicate dispatch [17, 31] modified to use external calling contexts. We also presented a sketch of our new Java extension based on that idea. In this paper, we follow that idea and present the design details of that language, which is now named *GluonJ*, its type system, and its implementation. Although we tried to emulate all the functionality of AspectJ in the previous paper (hence we proposed several predicates), the language presented in this paper is much simpler and shows better modularity and performance. Although GluonJ does not cover the full functionality of AOP languages, it shows that we can deal with crosscutting

structures to a certain degree within the confines of OOP; we do not need a different style of language construct dedicated to AOP.

The original predicate dispatch uses only variables visible in method scope, such as parameters and their fields, and thereby enables modular typechecking and compilation. Since GluonJ supports *contextual* predicate dispatch, in which a predicate refers to external calling contexts, like *who is a caller*, we cannot modularly typecheck and compile a GluonJ program. Note that crosscutting structures inherently show bad locality in the sense of traditional modularity and hence have difficulty in modular compilation. However, GluonJ programs are still *mostly* modular. This paper presents only limited parts of typechecking and compilation need global program analysis, from the viewpoints of not only practice but also theory. Another interesting issue is execution performance. This paper proposes an implementation technique, which involves negligible dispatch overhead. It also shows this fact through experiments.

In the rest, Section 2 briefly overviews functionality that typical AOP languages provide. Section 3 proposes GluonJ, which enables that functionality within OOP. Section 4 and 5 show the calculus and implementation of GluonJ. Section 6 mentions related work and Section 7 concludes this paper.

2 Objectives of AOP languages

A contribution of AOP would be putting a spotlight on crosscutting structures in software and practical demands on non-invasive modifications. These two might have been known before AOP but AOP has boosted comprehensive research on them.

2.1 Crosscutting structures

Software involves a number of crosscutting concerns, which are not only non-functional ones such as logging and transaction but also functional ones. This fact was revealed and reported by a number of AOP researchers at a series of AOSD conferences [3]. From the viewpoint of programming language design, several language constructs have been developed for joining components (or objects) in a crosscutting way. The most well-known construct is a pointcut-advice mechanism of AspectJ.

In AspectJ, a special method called *advice* can be implicitly invoked when another method is executed. It can be invoked also *in the middle of* a method body. This is a unique feature of AOP. Since it is considered as breaking the information hiding principle [36] on a method, its benefits have been controversial [41] and several ideas for dealing with this have been proposed so far [1, 20]. On the other hand, this feature is necessary to implement a number of crosscutting concerns represented by a famous logging concern. Figure 1 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. Although the Logging aspect is a simple example, the

```
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");
  }
}
```

Fig. 1. A logging aspect.

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 1 is executed.

2.2 Non-invasive modification

Another issue that AOP put a spotlight on is non-invasive (or transparent) extension. AOP enables software customization without invasively modifying the original source program due to the obliviousness property [18]. This level of customizability is not available by other extension mechanisms such as inheritance, mixins [8], mixin layers [39], family polymorphism [16], traits [38], and nested inheritance [33], although they also support non-invasive extension in some sense. 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 also have to modify all occurrences of object creation included in the original program to use the new version.

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

```
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());
  }
}
```

Fig. 2. A simple interpreter in AspectJ.

practical needs. A good example is JastAdd [15], 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.

Figure 2 shows a simplified example of compiler/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. Note that the aspect *destructively* extends 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.[3] The original program (Calc and AddExpr) remains the same; the behavior reverts to the original if the FloatExt aspect is removed.

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

While the extension by inheritance, mixin, and so on are useful for building libraries, the extension by AOP is useful for building frameworks, where the programs are reused as a whole and customized partly. Existing application frameworks written without AOP, therefore, come with a large amount of scaffolding code to enable the non-invasive reuse of a whole program within the confines of object-oriented programming. For example, Ruby on Rails [21] comes with a generator of scaffolding code. Dependency injection containers like Spring [19, 26] can be regarded as frameworks that provide scaffolding to customize composition of components. The Eclipse platform includes a number of interface types and extension points to develop plug-ins [43]. The plug-in mechanism of Eclipse is typical scaffolding; it checks the existence of plug-in components at every extension point at runtime and, if any, executes the components. The scaffolding code complicates the development of frameworks.

2.3 Language constructs

AOP languages such as AspectJ have been tackling the two issues above (not limited to them, though). Their approach has been to provide new dedicated language constructs, *i.e.* pointcut and advice, for dealing with the two issues while keeping classes and methods for other programming issues. Hyper/J [35] would not seem to provide such a new dedicated language construct since it allows programmers to describe a program in normal Java. However, how to *weave* multiple parts of a class must be described separately in a dedicated language, which is far from typical OOP languages like Java and hence is a sort of new dedicated language construct.

Although the crosscutting structures in Section 2.1 have not got much attention, the non-invasive modification in Section 2.2 has been studied also in the contexts of OOP. For example, MixJuice [24] and Feature-Oriented Programming [4] enable this as much as AOP languages. The partial classes of C# allow adding new members to an existing class. The category of Objective-C also does. On the other hand, these languages do not provide linguistic supports for dealing with the crosscutting structures. Some researchers of Feature-Oriented Programming suggest using AOP constructs such as pointcut and advice for the crosscutting structures while using their OOP-based construct, which is a destructive style of mixin, for the non-invasive modification [2].

3 GluonJ

Our goal is to naturally enhance OOP language constructs to support not only the non-invasive modification but also the crosscutting structure mentioned in the previous section. Existing OOP languages have not paid much attention to the crosscutting structure or they have been using language constructs borrowed from AOP languages for it. AOP languages like AspectJ use two kinds of language constructs: classes and methods for normal modularization and pointcut

```
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());
  }
}
```

Fig. 3. A revising class in GluonJ.

and advice for crosscutting modularization. Unlike them, we introduce AOP flavor into method dispatch, which is essential to OOP. Our approach is to enhance predicate dispatch to use external calling contexts. This section illustrates this approach by presenting our language named *GluonJ*.

3.1 Revising classes

GluonJ is an extension of Java. In GluonJ, a class can consist of multiple modules: a normal Java class and some revising classes. A revising class non-invasively modifies the definition of its target normal Java class as an aspect in AspectJ does. Figure 3 presents a revising class that performs the same modification that the FloatExt aspect in Figure 2 does. The revising class FloatExt may look like a subclass of AddExpr but it is part of the definition of the AddExpr class. Note that the class name is followed by not extends but revises. Since that revising class overrides the original implementation of the eval method described in the AddExpr class, 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.

A static method in a revising class overrides the corresponding static method declared in its target class. If the static method of the target class is called, the implementation of the method in the revising class is invoked. If that static method is called from the revising class, however, the implementation in the target class is invoked as in the standard Java. This is for allowing an overriding method in a revising class to invoke the original implementation.

Adding a new member
A revising class can also add a new method or field. If a method or a field in a revising class is not declared in the target class, it is added to the definition of the target class. A revising class is not allowed to declare a constructor. Only the default constructor implicitly declared is available. Figure 4 is an example of revising class adding a new member. It adds a tag field and a print method to the AddExpr class. Note that the initial value of tag is given in the declaration. No explicit constructor is declared.

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

Fig. 4. A revising class adding new members.

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

Fig. 5. A class using a method added by a revising class.

To refer to the added members from a class different from that revising class, a using declaration must be described. The added members are visible only within the source file including the using declaration of that revising class. Figure 5 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 revising class Printing. This restriction might seem to decrease the non-invasiveness of the modification by revising classes. The reader might think that it is like rewriting the class names in object-creation expressions and hence reusing a whole program would be made difficult. However, the restriction is not a problem. If an original program is self-contained, it never accesses newly added members since it was written before the revising class. Only the classes written with or after the revising class may 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. We will mention this again later.

Design details

A method declared in a revising class may be overridden by a method in a subclass of the target class. Suppose that both FloatExt in Figure 3 and a subclass PlusEqExpr of AddExpr declare eval methods. If the eval method is called on an PlusEqExpr object, then the invoked eval method is one declared in the PlusEqExpr class. The eval method in FloatExt is invoked if super.eval() is executed in PlusEqExpr.

A revising class follows a normal class with respect to the visibility rule. It can 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 revising class itself. Method overriding by a revising class also follows the rule of the standard Java. It cannot override a private method in the target

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

Fig. 6. A method with a predicate.

class. This rule restricts the ability for the non-invasive modification but we adopt this rule for the information hiding principle [36]. This rule will also ease the fragile pointcut problem of AOP [28, 30, 42]. 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.

3.2 A within method

Method overriding by a revising class corresponds to advice with the execution pointcut in AspectJ. It can change the behavior of an existing method. On the other hand, AspectJ provides a richer pointcut language for dealing with crosscutting structures.

GluonJ uses enhanced predicate dispatch for providing a similar function to the within and withincode pointcuts of AspectJ. A revising class can declare a method with a predicate within, which we below call *a within method*. This method is selected for invocation only when its predicate is true. Otherwise, the overridden method in the target class is selected. 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 revising class Tracer adds a logging function as the Logging aspect in Figure 1 does. The recordVariable method in this revising class is invoked only when it is called on an Env object from the eval method in the VariableDecl class. It is invoked even if the static type of the Env object at the caller site is a super type of Env. Unlike the call pointcut of AspectJ, the apparent type of the receiver does not matter; the actual type is considered.

Roughly to say, a within method of GluonJ corresponds to a combination of call, within (or withincode), and target pointcuts in AspectJ. Since the call pointcut selects join points when a specified method is called on an object with a specified *static* type, the target pointcut is necessary to select join points by considering the dynamic type of the receiver object.

AspectJ programmers might misunderstand that a call pointcut selects method-call expressions at *caller* sites and hence it attaches advice in the middle of a method body of the caller, which would look like a violation of encapsulation. On the other hand, GluonJ provides more straightforward intuition. A within method gives a chance to replace the behavior of a method when the method is called from a specific caller site.

Design details
A within method must be declared in a revising class. It does not have to override a method in the target class or a super class of the target class. If it does not override the method, since it is a method newly added by a revising class, the source file of a method calling that within method must include a using declaration of that revising class. The caller method must satisfy the within predicate. Otherwise, it causes a type error.

Furthermore, in the current specification of the language, if a revising class declares a method foo with a within predicate, then it cannot declare another method foo with the same signature with/without a predicate. To declare more than one within method for foo, multiple classes revising the same class must be declared and each of them must declares one within method. As in Java, a revising class can declare multiple methods foo if they have different signatures.

A within method is overridden by a normal method with the same name and signature in a subclass of the target class. Suppose that a subclass ExEnv of Env declares another recordVariable method. Then the within method in Tracer in Figure 6 is not invoked when the recordVariable method is called on an ExEnv object from the env method.

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 [31], 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. JPred ensures two properties, exhaustiveness and unambiguity, by modular static typechecking. Exhaustiveness checking ensures that a program will not cause no-such-method errors at runtime. Unambiguity checking ensures that every method call in a program has a unique most-applicable method.

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 inherently depends on the externals. This fact removes some locality from methods and complicates modular compilation. However, as we show later, the compilation of a GluonJ program by our compiler is still mostly modular.[4] The exhaustiveness property is statically typechecked per source-file basis since our predicates are simple. The unambiguity property is guaranteed by the precedence order among revising classes, which is shown next. Only the correctness of method overriding and (part of) code generation need global typechecking and analysis.

[4] The unmodularity mainly comes from revising classes. If GluonJ did not provide a revising class but only allowed a within method declared in a normal class and we did not care about execution performance, a within method could be modularly typechecked and compiled.

```
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());
  }
}
```

Fig. 7. A requires clause.

3.3 Precedence order among revising classes

Since GluonJ allows multiple revising classes to revise the same normal class, it has a rule for resolving ambiguity among methods in those revising classes. Suppose that two revising classes R and S revise the same class C and they declare methods m with the same name and signature. When m is called on an instance of C, GluonJ selects the m method declared in the revising class with higher precedence. This is also true for within methods.

The precedence order among revising classes is specified by requires clauses in class declarations. Figure 7 shows an example. This revising class StringExt declares that it requires another revising class FloatExt in Figure 3. This means that StringExt has higher precedence than FloatExt while both StringExt and FloatExt revise the AddExpr class. Thus, a call to eval on an AddExpr object invokes the eval method declared in StringExt. The eval method of StringExt "overrides" eval of FloatExt, which then "overrides" eval of AddExpr. super.eval() calls the overridden method of the class with the next precedence.

A requires clause can include multiple revising class names separated by comma. The left has higher precedence. Moreover, a revising class may require another revising class that revises a different class than the former one does. In a valid GluonJ program, the requires relation over the classes revising the same class must be a total order. This constraint ensures that the precedence order does not have ambiguity and hence the most specific method is uniquely determined among methods of revising classes.

Notes on within methods
If multiple within methods override the same normal method and their within predicates specify the same location, the most specific one selected for invocation is the within method declared in the revising class with the highest precedence. The within predicates may have an overlap. For example, they may be within C and within C.foo(). Both of them match the location of the foo method in the class C. Whether a within predicate specifies a class name or a method name does not affect the selection of the most specific method. The most specific method is the method declared in the revising class with the highest precedence among normal methods and the methods with predicates matching the caller location.

When a within method calls a method on super, the location of that super call is within the body of the method including that super call. Suppose that multiple within methods override the same method m and their within predicates specify the same class C. When the method m is called from the class C, GluonJ invokes the within method with the highest precedence. Then, if this within method executes super.m(), since the caller is not C any longer but the revising class, the next invoked method is not the within method with the second highest precedence. It is the original method m overridden by the within methods. In Figure 6, the within method recordVariable in Tracer is invoked when it is called from the eval method in VariableDecl. Then, this within method in Tracer executes super.recordVariable(name, value). Here, the caller is not the eval method but this within method in Tracer.

4 Core calculus of GluonJ and its type soundness

In this section, we give a formal calculus called GluonFJ as an extension of Featherweight Java (FJ) [25] to discuss modular typechecking and type soundness. GluonFJ adds revising classes and within methods to FJ. For simplicity, we do not model super calls, and fields of revising classes; within clauses are restricted so that only class names can be given as source code locations.

We first give the syntax, typing rules, and operational semantics of GluonFJ and then prove type soundness.

4.1 Syntax

As mentioned above, GluonFJ has revising classes 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` $\overline{\texttt{R}}$ `{` $\overline{\texttt{C}}$ $\overline{\texttt{f}}$`;` $\overline{\texttt{M}}$ `}`	normal classes
	|	`class R revises C using` $\overline{\texttt{R}}$ `{` $\overline{\texttt{M}}$ `}`	revising classes
L	::=	C | R	locations
M	::=	C `m(`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`){ return e; }` [`within` L]	methods
e	::=	x | `e.f` | `e.m(`$\overline{\texttt{e}}$`)` | `new C(`$\overline{\texttt{e}}$`)` | `e in` L	expressions
v, w	::=	`new C(`$\overline{\texttt{v}}$`)`	values

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 normal class names; R ranges over revising class 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 normal class names and revising class names are disjoint and the (denumerable) set of revising class 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 revising class names $\overline{\texttt{R}}$ is sorted according to this order.

`CL` is a normal (or revising) class definition, consisting of its name, a super class name (or the class name that it revises, respectively), revising class names that it uses, fields, and methods. A revising class 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 normal 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 normal/revising class 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 $\overline{\texttt{e}}$ for variables $\overline{\texttt{x}}$ by $[\overline{\texttt{e}}/\overline{\texttt{x}}]$.

A GluonFJ program is a triple consisting of a class table CT, which is a mapping from normal class names to normal class definitions, a revising class table RT, which is also a mapping from revising class names to revising class 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 $CT(\texttt{C}) = \texttt{class C extends D } \cdots \texttt{ \{} \cdots \texttt{\}}$. Similarly, we use `R rev C` and `L using` $\overline{\texttt{R}}$.

Finally, we always assume fixed class tables, which are assumed to satisfy the following sanity conditions: (1) $CT(\texttt{C}) = \texttt{class C } \cdots$ for every $\texttt{C} \in dom(CT)$ and similarly for RT; (2) $\texttt{Object} \notin dom(CT) \cup dom(RT)$; (3) for every class name `L` (except `Object`) appearing anywhere in CT and RT, we have $\texttt{L} \in dom(CT) \cup dom(RT)$; 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 $\mathit{fields}(\texttt{C})$ returns all the fields of `C` and its super classes with their types as $\overline{\texttt{C}}\ \overline{\texttt{f}}$. It is defined by the following rules:

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \text{(F-Object)}$$

$$\frac{\texttt{class C extends D using } \overline{\texttt{R}} \texttt{ \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; } \overline{\texttt{M}} \texttt{ \}} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}} \qquad \text{(F-Class)}$$

The function $\mathit{lowest}(\texttt{C}, \overline{\texttt{R}})$ returns the first revising class that revises `C`, found in $\overline{\texttt{R}}$, or just returns `C` if there is no such revising class.

$$\mathit{lowest}(\texttt{C}, \overline{\texttt{R}}) = \begin{cases} \texttt{R}_j \text{ if } \texttt{R}_j \texttt{ rev C}, \text{ and } \neg\exists k < j.\texttt{R}_k \texttt{ rev C} \\ \texttt{C} \text{ if } \neg\exists j.\texttt{R}_j \texttt{ rev C} \end{cases}$$

The function $super(\mathtt{L}, \overline{\mathtt{R}})$ returns the next class of L to look up and is defined by:

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

(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 $dom(RT)$.

$$lowest(\mathtt{C}) = lowest(\mathtt{C}, dom(RT))$$
$$super(\mathtt{L}) = super(\mathtt{L}, dom(RT))$$

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:

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

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

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 $\mathtt{m}\ \cdots\ \mathtt{within\ L}'$ in $\overline{\mathtt{M}}$); then, the signature is equivalent to that from the next class, represented by $super(\mathtt{L}, \overline{\mathtt{R}})$. 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 revising classes. 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*:

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

$$\frac{\texttt{class L \{extends,revises\} C using } \overline{\texttt{R}} \texttt{ \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; } \overline{\texttt{M}} \texttt{ \}} \quad \texttt{m [within L']} \notin \overline{\texttt{M}} \qquad \mathit{mbody}(\texttt{m}, \mathit{super}(\texttt{L}), \texttt{L}') = \overline{\texttt{x}}\texttt{.e in L''}}{\mathit{mbody}(\texttt{m}, \texttt{L}, \texttt{L}') = \overline{\texttt{x}}\texttt{.e in L''}} \quad \text{(MB-Super)}$$

Unlike *mtype*, however, it (implicitly) uses all revising classes (remember that $\mathit{super}(\texttt{L})$ is shorthand for $\mathit{super}(\texttt{L}, \mathit{dom}(RT))$).

4.3 Type System

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

$$\texttt{C} <: \texttt{C} \quad \text{(S-Refl)}$$

$$\frac{\texttt{C} <: \texttt{D} \qquad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}} \quad \text{(S-Trans)}$$

$$\frac{\texttt{C ext D}}{\texttt{C} <: \texttt{D}} \quad \text{(S-Extends)}$$

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

$$\texttt{L}; \Gamma \vdash \texttt{x} : \Gamma(\texttt{x}) \quad \text{(T-Var)}$$

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

$$\frac{\texttt{L using } \overline{\texttt{R}} \qquad \begin{array}{c}\texttt{L}; \Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \qquad \texttt{L}; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{C}} \\ \mathit{mtype}(\texttt{m}, \mathit{lowest}(\texttt{C}_0, \overline{\texttt{R}}), \overline{\texttt{R}}, \texttt{L}) = \overline{\texttt{D}} \to \texttt{C} \qquad \overline{\texttt{C}} <: \overline{\texttt{D}}\end{array}}{\texttt{L}; \Gamma \vdash \texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)} : \texttt{C}} \quad \text{(T-Invk)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}} \\ \texttt{L}; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{C}} \qquad \overline{\texttt{C}} <: \overline{\texttt{D}}\end{array}}{\texttt{L}; \Gamma \vdash \texttt{new C(}\overline{\texttt{e}}\texttt{)} : \texttt{C}} \quad \text{(T-New)}$$

$$\frac{\texttt{L}'; \Gamma \vdash \texttt{e} : \texttt{C}}{\texttt{L}; \Gamma \vdash \texttt{e in L}' : \texttt{C}} \quad \text{(T-In)}$$

Most rules are straightforward adaptations from FJ typing rules. In the rule T-Invk, method lookup starts from $\mathit{lowest}(\texttt{C}, \overline{\texttt{R}})$, 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 $\mathtt{L}'$.

The type judgment for methods is of the form `M OK IN L`, read "method `M` is well typed in class `L`." The typing rules are given below:

$$\frac{\mathtt{C};\overline{\mathtt{x}}:\overline{\mathtt{C}},\mathtt{this}:\mathtt{C}\vdash \mathtt{e}_0:\mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ \text{for any } \mathtt{L}, \text{ if } \mathit{mtype}(\mathtt{m}, \mathit{super}(\mathtt{C}), \mathit{dom}(RT), \mathtt{L}) = \overline{\mathtt{D}}\to\mathtt{D}_0, \text{ then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{OK\ IN}\ \mathtt{C}} \quad \text{(T-Method)}$$

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

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 revising class) 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 revising classes 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." The typing rules, which are straightforward, are given as follows:

$$\frac{\overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C}}{\mathtt{class\ C\ extends\ D\ using}\ \overline{\mathtt{R}}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \quad \text{(T-Class)}$$

$$\frac{\overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{R}}{\mathtt{class\ R\ revises\ D\ using}\ \overline{\mathtt{R}}\ \{\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \quad \text{(T-RClass)}$$

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 $\mathtt{e}'$ in one step in `L`." Reduction rules are given below:

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

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

$$\frac{\mathit{mbody}(\mathtt{m}, \mathit{lowest}(\mathtt{C}), \mathtt{L}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ L'}}{\mathtt{L} \vdash \mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{w}}) \longrightarrow ([\overline{\mathtt{w}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{v}})/\mathtt{this}]\mathtt{e}_0)\ \mathtt{in\ L'}} \quad \text{(R-Invk)}$$

All rules are straightforward. When a method is invoked on an object of C, method lookup starts from *lowest*(C), the first revising class 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. 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 could formalize the call-by-name semantics, as in FJ, by annotating expressions with in L before substitution.)

We show congruence rules below:

$$\frac{\mathtt{L} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{L} \vdash \mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0{}'.\mathtt{f}} \quad \text{(RC-Field)}$$

$$\frac{\mathtt{L} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{L} \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0{}'.\mathtt{m}(\overline{\mathtt{e}})} \quad \text{(RC-Invk-Recv)}$$

$$\frac{\mathtt{L} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{L} \vdash \mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i{}',\ldots)} \quad \text{(RC-Invk-Arg)}$$

$$\frac{\mathtt{L} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{L} \vdash \mathtt{new\ C}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{new\ C}(\ldots,\mathtt{e}_i{}',\ldots)} \quad \text{(RC-New-Arg)}$$

$$\frac{\mathtt{L}' \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{L} \vdash \mathtt{e}_0\ \mathtt{in\ L'} \longrightarrow \mathtt{e}_0\ \mathtt{in\ L'}} \quad \text{(RC-In)}$$

These rules represent that the order of argument evaluation is not really fixed.

4.5 Type Soundness

GluonFJ is type sound; it enjoys subject reduction and progress [46]. Here, we suppose that (CT, RT) OK and write $\mathtt{L} \vdash \mathtt{e}_1 \longrightarrow^* \mathtt{e}_n$ when $\mathtt{L} \vdash \mathtt{e}_1 \longrightarrow \mathtt{e}_2$, ..., $\mathtt{L} \vdash \mathtt{e}_{n-1} \longrightarrow \mathtt{e}_n$.

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* $\mathtt{e}'$ *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* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is either a value* v *with* $\mathtt{L}; \emptyset \vdash \mathtt{v} : \mathtt{D}$ *for some* $\mathtt{D} <: \mathtt{C}$.

Proof. Immediate from Theorems 1 and 2.

5 Implementation

We implement a revising class by translating it into a subclass of its target class and then adjusting a whole program so that the revising class will be used instead of its target class in instantiations.

5.1 Compilation Overview

The compilation of a GluonJ program is mostly modular. It consists of two stages: source-to-bytecode translation and linking. We implemented the translation stage by extending JastAddJ [15] and the linking stage by using Javassist [9]. The linking stage is bytecode transformation and it is executed at load time or statically at the end of compile time.

Translation stage
At the translation stage, a source file is separately compiled into Java bytecode. This compilation is modular; it needs only the revising classes specified by using declarations 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 revising class is a direct subclass of its target class. It also translates a member-access expression if it accesses a member added by a revising class. 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 revising class Printing given by a using declaration (as in Figure 5). A within method is compiled similarly to a normal method. The within predicate is translated into Java annotations to that method.

A constructor of a revising class is compiled differently from normal constructors of Java classes. Recall that a revising class 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 revising class 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 revising classes 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 individually to a normal class if only all the revising classes are given, this stage is still fairly modular. It never refers to the set of all normal classes.

Our linker first computes the precedence order among the revising classes and checks its validity. Then it sets the direct super class of every revising class to its target class. If multiple classes 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 revising

class 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 revising class. 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 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 revising classes is valid. Suppose that two revising classes revise the same class. If these revising classes declare methods with the same name and parameter types but with different (not-covariant) return types, then this method overriding is invalid since the revising classes now have an inheritance relationship.[5]

The linker also generates constructors for every revising class 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 revising class FloatExt in Figure 3 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 revising class.

Finally, our linker applies bytecode transformation to every class file. If a normal class C is revised by a revising class 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 revising class with the highest precedence, that is, the lowest subclass. Furthermore, if a static method is overridden by a revising class, then all occurrences of the call to that static method is redirected to the overriding method in the revising class.

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. This optimization, however, requires global program analysis and transformation.

[5] Actually, most of these checks could be performed by the compiler. If classes that revise the same class are totally ordered by requires clauses, it suffices to check the revising class at the bottom against T-MethodR. So, if the compiler checks each class for valid method overriding using only the classes it (transitively) requires, then the linker only has to check if requires clauses form a total order.

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 revising class 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 ? 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.

5.2 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 revising class table is empty, as the target language. Precisely speaking, we need type casts `(C)e` in the target language. Typing rules and reduction rules are given as follows:

$$\frac{\mathtt{L};\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\mathtt{L};\Gamma \vdash \mathtt{(C)e}_0 : \mathtt{C}} \qquad \text{(T-UCast)}$$

$$\frac{\mathtt{L};\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\mathtt{L};\Gamma \vdash \mathtt{(C)e}_0 : \mathtt{C}} \qquad \text{(T-DCast)}$$

$$\frac{\texttt{C} <: \texttt{D}}{\texttt{L} \vdash \texttt{(D)(new C}(\overline{\texttt{v}})\texttt{)} \longrightarrow \texttt{new C}(\overline{\texttt{v}})} \qquad \text{(R-Cast)}$$

$$\frac{\texttt{L} \vdash \texttt{e} \longrightarrow \texttt{e}'}{\texttt{L} \vdash \texttt{(C)e} \longrightarrow \texttt{(C)e}'} \qquad \text{(RC-Cast)}$$

(Since we conjecture that casts inserted through compilation do not fail, we intentionally omit the typing rule for so-called stupid casts [25], which represent casts that have failed during execution.) 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 $\mathit{origin}(\texttt{m}, \texttt{L})$ returns the name of the highest super class of L in which m (be it normal or within) is defined.

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

$$\frac{\mathit{origin}(\texttt{m}, \mathit{super}(\texttt{L})) = \texttt{L}'}{\mathit{origin}(\texttt{m}, \texttt{L}) = \texttt{L}'}$$

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

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

The function $\mathit{within}(\texttt{m}, \texttt{L})$, which collects all locations used as predicates for m in L, is defined to be $\mathit{within}_{sub}(\texttt{m}, \mathit{thistype}(\mathit{origin}(\texttt{m}, \texttt{L})))$ where $\mathit{thistype}(\texttt{C}) = \texttt{C}$ and $\mathit{thistype}(\texttt{R}) = \texttt{C}$ 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, $\mathit{within}_{sub}(\texttt{m}, \texttt{B}) = \{\texttt{D}\}$ and $\mathit{within}(\texttt{m}, \texttt{B}) = \mathit{within}(\texttt{m}, \texttt{R2}) = \mathit{within}(\texttt{m}, \texttt{R1}) = \mathit{within}(\texttt{m}, \texttt{A}) = \{\texttt{C}, \texttt{D}\}$. When $\texttt{L} \in \mathit{within}(\texttt{m}, \texttt{C})$ the invocation of m on C from class L will be translated to the invocation of $\texttt{m}_\texttt{L}$, which stands for a name mangled from m and L.

The judgment for translation is of the form $\texttt{L}; \Gamma \vdash \texttt{e} \Longrightarrow \texttt{e}'$, read "expression e is translated to $\texttt{e}'$ under L and Γ." Translation rules are given below.

The rules to translate variables and field accesses are trivial.

$$\frac{}{\mathtt{L};\Gamma \vdash \mathtt{x} \Longrightarrow \mathtt{x}} \quad \text{(Tr-Var)}$$

$$\frac{\mathtt{L};\Gamma \vdash \mathtt{e}_0 \Longrightarrow \mathtt{e'}_0}{\mathtt{L};\Gamma \vdash \mathtt{e}_0\mathtt{.f} \Longrightarrow \mathtt{e'}_0\mathtt{.f}} \quad \text{(Tr-Field)}$$

In translating a method invocation, we ensure that a method only available in a revising class can be invoked, by casting the receiver to $\mathtt{L'}$, which is the lowest revising class that revises the receiver type. As we mentioned above, the method name may be changed if the caller location $\mathtt{L}$ is found in $\mathit{within}(\mathtt{m}, \mathtt{C}_0)$.

$$\frac{\mathtt{L};\Gamma \vdash \mathtt{e}_0 \Longrightarrow \mathtt{e_0}' \qquad \mathtt{L};\Gamma \vdash \overline{\mathtt{e}} \Longrightarrow \overline{\mathtt{e}}' \qquad \mathtt{L};\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathtt{L}\ \mathtt{using}\ \overline{\mathtt{R}} \qquad \mathit{lowest}(\mathtt{C}_0, \overline{\mathtt{R}}) = \mathtt{L'} \qquad \mathtt{m'} = \begin{cases} \mathtt{m_L} \text{ if } \mathtt{L} \in \mathit{within}(\mathtt{m}, \mathtt{C}_0) \\ \mathtt{m} \quad \text{otherwise} \end{cases}}{\mathtt{L};\Gamma \vdash \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \Longrightarrow \mathtt{((L')e_0')}\mathtt{.m'}(\overline{\mathtt{e}}')} \quad \text{(Tr-Inv)}$$

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

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

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 (see the rule T-Method in the last section). 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 $\mathit{within}(\mathtt{m}, \mathtt{L})$.

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

For example, the method `m` in class `B` above will translate to three methods named `m`, $\mathtt{m_D}$, and $\mathtt{m_C}$, even though no class revising `B` has a method with within

C. (B will have R1, which has $\mathtt{m_C}$ after translation, as a super class of B.) The translation of a within method is straightforward.

$$\frac{\mathtt{L};\overline{\mathtt{x}}{:}\overline{\mathtt{B}},\mathtt{this}:\mathit{thistype}(\mathtt{L}) \vdash \mathtt{e} \Longrightarrow \mathtt{e'}}{\mathtt{L} \vdash \mathtt{B\ m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\mathtt{\{\ return\ e;\ \}\ within\ L'} \Longrightarrow \mathtt{B}\ \mathtt{m_{L'}}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\mathtt{\{\ return\ e';\ \}}} \quad \text{(Tr-Within)}$$

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

$$\frac{\mathit{super}(\mathtt{L}) = \mathtt{L'} \qquad \mathtt{L} \vdash \overline{\mathtt{M}} \Longrightarrow \overline{\mathtt{M}}'}{\begin{array}{l}\vdash \mathtt{class\ L\ \{extends, revises\}\ D\ using}\ \overline{\mathtt{R}}\ \mathtt{\{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}} \\ \qquad \Longrightarrow \mathtt{class\ L\ extends\ L'\ \{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}'\ \mathtt{\}}\end{array}} \quad \text{(Tr-Class)}$$

The super class D is replaced with L′, which is the name of the next class when looking up definitions. Note that every class 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 normal and revising class 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 $\mathtt{R} \in \mathit{dom}(RT)$.

5.3 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 $\mathtt{new\ C(\overline{e})}$, in which case it will become $\mathit{lowest}(\mathtt{C})$. Since we mention two programs before and after translation at the same time, we explicitly say 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* $\mathit{lowest}(\mathtt{C})$.

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

Theorem 4. *If* (CT, RT) *is OK, then there exists* CT' *such that* 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.

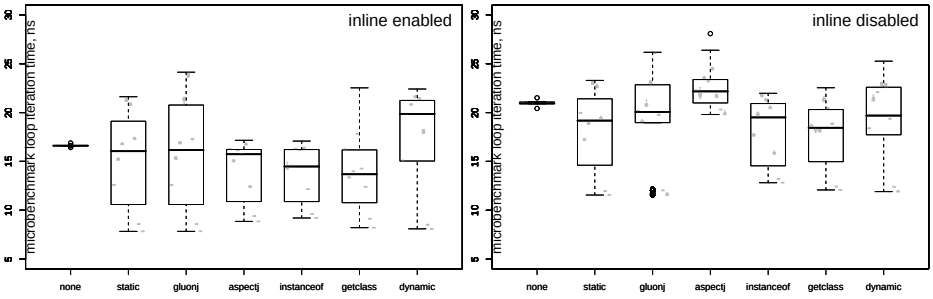

Fig. 8. Overheads of method dispatch techniques.

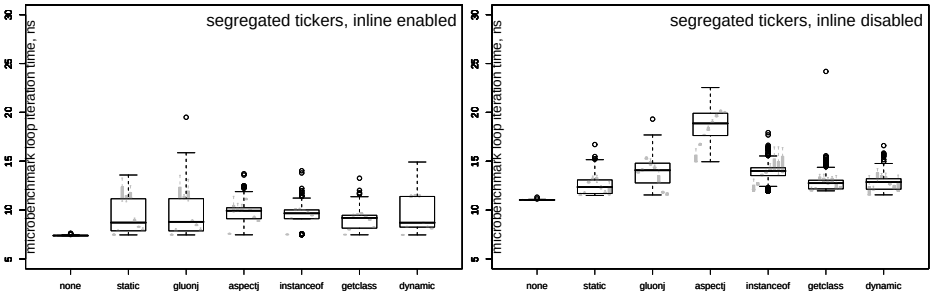

Fig. 9. `callers` and `tickers` arrays segregated by object type.

5.4 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). Details of the experiments will be presented in our technical report [11].

5.5 Microbenchmark

We implemented a microbenchmark incorporating several implementation techniques that can be used to alternate method dispatch. Since microbenchmark executes a tight loop, it allows us to estimate impact of different dispatch techniques on performance of a hot loop, compiled by the JIT compiler with highest optimization level.

Microbenchmark constructs two sets of objects: callers and callees. Caller object has a method call, which receives callee object as a parameter, and invokes method tick on callee (called Ticker in the microbenchmark). The microbenchmark operation proceeds by sequentially taking one object from each array and

```
for (int i = 0; i < n; i++) {
  callers[(count1+count2) % callers.length]
    .work(tickers[(count1+count2) % tickers.length]);
}

// regular caller, always calls tick()
class Caller {
  public void work(Ticker ticker) {
    ticker.tick();
  }
}

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

class Ticker3 extends Ticker2 {
  public void tick() {
    Microbench.count1 += 1;
  }
}
```

Fig. 10. Microbenchmark workload

calling method call on caller object, passing callee object as a parameter. The operation is repeated 10 million times in a hot loop. 200 thousands untimed iterations are performed for warm-up.

Callee objects have two versions of the target method (tick and tick2) Objects, implementing dispatch to alternative method version are injected into caller and callee arrays in varying quantities, making it possible to control the ratio of dispatching the call to original and alternate target method.

We found in initial experiments, that uniform or close to uniform distribution of object types results in JIT compiler using additional optimizations, so the comparison becomes biased towards cases with less object diversity. In order to reduce bias, callee objects of three types are used: Ticker and Ticker3 present *original* functionality, while Ticker2 is emulation of a within method. Figure 10 presents Ticker and Ticker3. Results for two series of experiments are presented, when objects of different type are segregated, and when objects are shuffled pseudo-randomly. In each series two experiments were performed, with JIT compiler inlining enabled and disabled.

Figure 8 and 9 show results of experiments. Dispatch techniques are on the x axis, and duration of a single iteration of microbenchmark in ns is shown on y axis. Gray dots visible on the graphs shows the data points of performance measurement with particular ratio between original call target tick and modified

```
// statically altered dispatch, always calls tick2()
class Caller2Static extends Caller {
  public void work(Ticker ticker) {
    ticker.tick2();
  }
}

class Ticker {
  // for static caller-side dispatch
  public void tick2() {
    // copy of tick() code
    Microbench.count1 += 1;
  }
}

class Ticker2 extends Ticker {
  public void tick2() {
    Microbench.count2 += 1;
  }
}
```

Fig. 11. Static dispatch

call target tick2. The leftmost column none presents performance of the case when no modification to dispatch were made, and serves as a reference point. Note that this case also exibits dependency on uniformity of calls.

While experimenting with the microbenchmark we saw that number of potential call targets and particular ordering of calls to different target methods can have significant impact on performance (*e.g.* compare the left graphs of Figure 8 and 9). However, most of optimizations that are applied by the JIT compiler to our microbenchmarks are also likely to be possible with our within method used in practice. For this reason, we consider microbenchmark results relevant.

When optimizations of the call site are possible, all dispatch methods exhibit similar performance. However, different techniques may result in different profile of the call sites, which in turn may prevent JIT from applying best possible optimization. In particular, static techniques shown in Figure 11 result in a single multiple-target call site, which may or may not be subject to guarded devirtualization and inlining optimizations. If the diversity is low, and single target is taken, the performance is close to the best possible, but if the call site in fact dispatches to multiple different implementations, then performance degrades significantly. The techniques adopted in GluonJ implementation is essentially the same as static techniques excpet that tick2 in Ticker delegates to tick .

This contrasts with techniques instanceof or its slight variation getclass, which use conditional construct, and two call sites, each having less variation in call target (Figure 12 and 13). This results in consistent ability of JIT compiler to optimize call sequence.

```
// static caller with instanceof check
class Caller2Instanceof extends Caller {
  public void work (Ticker ticker) {
    if (ticker instanceof Ticker2 && !(ticker instanceof Ticker3))
      ticker.tick2();
    else
      ticker.tick();
  }
}
```

Fig. 12. Dispatch via instanceof check

```
class Caller2Getclass extends Caller {
  public void work (Ticker ticker) {
    if (ticker.getClass() == Ticker2.class)
      ticker.tick2();
    else
      ticker.tick();
  }
}
```

Fig. 13. Dispatch via getclass check

dynamic dispatch techniques shown in Figure 14 delegates the dispatch decision to callee side, which also proved to be a challenge to the JIT compiler. Its performance is somewhat worse than fastest techniques.

For the sake of comparison, we also implemented the same microbenchmark using AspectJ as in Figure 15. The resulting code is similar to our instanceof techniques, though distribution of code between classes is different. Performance of code produced by AspectJ is similar to other techniques if inlining is enabled, and is somewhat slower if inlining is disabled.

Since the variation of microbenchmark performance due to applicability of agressive JIT optimization has the order of magnitude of of 2–3 times, and variation between performance of different dispatch techniques is much less, we expect that choice of implementation techniques for non-invasive modification will not have visible impact on hot loop performance.

5.6 the DaCapo benchmarks

Next, we used the DaCapo suite of benchmarks [6] to evaluate potential impact of within methods on real applications, especially in the cold code. To conservatively estimate the impact, we instrumented all application methods with bytecode, 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 the different implementation techniques mentioned above. Figure 16

```
// marker caller class for dynamic callee-dispatch
class Caller2Dynamic extends Caller {
  public void work(Ticker ticker) {
    ticker.tick(Caller2Dynamic.class);
  }
}

// different caller class for callee-dispatch
class CallerDynamic extends Caller {
  public void work(Ticker ticker) {
    ticker.tick(CallerDynamic.class);
  }
}

class Ticker {
  public void tick(Class x) {
    if (x == Caller2Dynamic.class)
      Microbench.count2 += 1;
    else
      Microbench.count1 += 1;
  }
}
```

Fig. 14. Dispatch via dynamic check in the callee

shows the results on 8 of 11 DaCapo benchmarks (the other benchmarks did not work after the instrumentation). Overhead of the instrumentation on the first iteration is up to 22% for well-behaving benchmarks, and up to 151% for `luindex`. 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. For the other benchmarks, the impact of any implementation technique is negligible on the 10th iteration. Table 1 lists the overheads on each benchmark. The details of the bytecode transformation are shown in Figure 17 to 20.

6 Related work

A large number of languages and systems must be cited as related work. We have already mentioned some of them. We below show other related work but it does not cover all.

Non-invasive modification has been actively studied. Classboxes [5] and Context-Oriented Programming (COP) [23] allow applying new behavior to all instances in an original program although the programmers have to explicitly specify when the new behavior gets effective. While they allow dynamically switching old and new behavior for one instance, an original program must be slightly modified if

```
aspect TickerR {
  pointcut tick(): call(void Ticker.tick());
  pointcut withinCaller(): within(Caller2);
  void around():
    tick() && withinCaller() && target(Ticker2) && !target(Ticker3) {
    final_tick2();
  }

  private final void final_tick2() {
    Microbench.count2 += 1;
  }
}
```

Fig. 15. Implementation using AspectJ

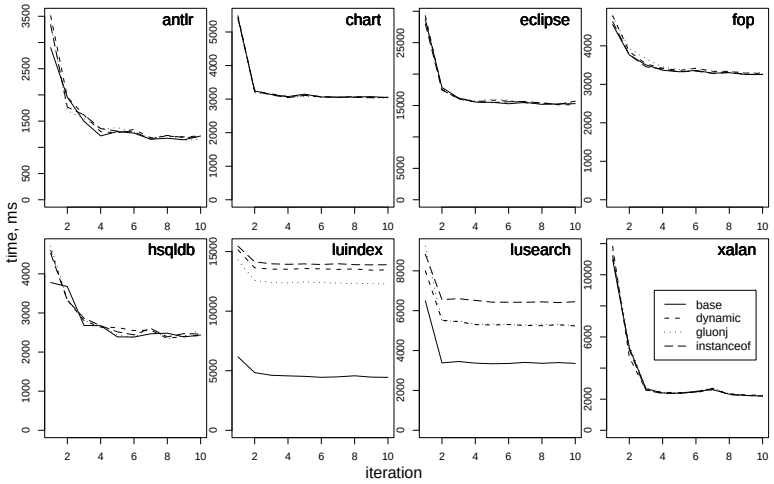

Fig. 16. DaCapo benchmarks.

it needs to activate new behavior. This is good with respect to modularity but not suitable when the program is a framework and programmers want to reuse the whole program after customization. Classboxes activate new behavior only within a new package. In COP, a layer must be explicitly activated for using new behavior. Their method dispatch can be regarded as also using external calling contexts like GluonJ. The proposed implementation of classboxes needs runtime stack inspection, which will imply non-negligible overheads.

On the other hand, our GluonJ is categorized into the non-invasiveness in which only the new behavior is always effective. JavaGI [45] belongs to this category. Open classes [13] and expanders [44] also belong 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. [29] allows dynamically changing class definitions. A difference from those

```
public class Hello {
  void x() { y(); }
  void y() {}
}
```

Fig. 17. Sample class

```
public class Hello {
  void x() {
    Hello hello = this;
    hello.y$prime();
  }

  void y() {}
  void x$prime() { x(); }
  void y$prime() { y(); }
}
```

Fig. 18. Imitation of `static` dispatch

```
public class Hello {
  void x() {
    Hello hello = this;
    if (hello instanceof Hello) {
      hello.y();
      Object obj = null;
    } else {
      hello.y();
      Object obj1 = null;
    }
  }

  void y() {}
}
```

Fig. 19. Imitation of caller-side dispatch with `instanceof` check

```
public class Hello {
  void x() {
    Hello hello = this;
    hello.y$prime(Hello);
  }

  void y() {}

  void x$prime(Class class1) {
    Hello hello = this;
    hello.y$prime(Hello);
  }

  void y$prime(Class class1) {}
}
```

Fig. 20. Imitation of `dynamic` dispatch by callee

Table 1. Impact of bytecode transformation on DaCapo benchmarks

	Overhead for iteration	
benchmark	first	tenth
antlr	4 to 19%	-2 to -3%
chart	-1 to -2%	0%
eclipse	1 to 3%	1 to 7%
fop	8 to 12%	1%
hsqldb	17 to 22%	1 to 6%
luindex	122 to 151%	178 to 214%
lusearch	26 to 37%	61 to 97%
xalan	3 to 6%	2 to 4%

work is that GluonJ also provides support for crosscutting structures by using the idea of predicate dispatch.

J&$_s$ [37] also supports non-invasive modification 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 typechecing point of view, the view change operation plays a role somewhat similar to using clauses, but the execution model is rather different since new members do not always override old ones.

AspectJ2EE [14] 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 the call pointcut. Hence it does not provide the same expressiveness that GlounJ does.

Presenting similarity between predicate dispatch and the pointcut-advice of AOP is not new. This idea has been pointed out by other researchers [34, 7, 22]. The idea of method dispatch depending on a caller object is also found in [40]. Our contribution is that we proposed an OOP language based on this idea and discussed modular typechecking and compilation.

We have been developing a series of AOP languages and some of the languages inherited the name GluonJ. However, the design of those languages are different. The first GluonJ published in [12] 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 [32] 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.

7 Conclusion

We presented GluonJ, which supports revising classes and within methods. These language constructs are natural enhancement to method dispatch and help non-invasive modification and composition of crosscutting structures. Helping not only the first one but also the second one is unique among method dispatch

mechanisms. We also showed that typechecking of GluonJ programs is modular except checking T-MethodR through the calculus. Compilation is also mostly modular. Only the bytecode transformation lastly applied to each class requires all the revising classes. Furthermore, we mentioned that the execution performance is negligible through experiments. GluonJ does not cover full functionality of typical AOP languages. Extending our approach to cover a wider range of it is our future work.

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. Aspect-Oriented Software Association: Int'l Conf. on Aspect-Oriented Software Development. http://www.aosd.net
4. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2004)
5. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the scope of change in Java. In: Proc. of ACM OOPSLA. pp. 177–189 (2005)
6. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proc. of ACM OOPSLA. pp. 169–190. ACM (2006)
7. Bockisch, C., Haupt, M., Mezini, M.: Dynamic virtual join point dispatch. Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '06) (2006)
8. Bracha, G., Cook, W.: Mixin-based inheritance. In: Proc. of OOPSLA/ECOOP '90. pp. 303–311. ACM Press (1990)
9. Chiba, S.: Load-time structural reflection in Java. In: ECOOP 2000. pp. 313–336. LNCS 1850, Springer-Verlag (2000)
10. 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)
11. Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular composition of crosscutting structures by contextual predicate dispatch (extended version). Research Reports C-267, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology (December 2009)
12. Chiba, S., Ishikawa, R.: Aspect-oriented programming beyond dependency injection. In: ECOOP 2005. pp. 121–143. LNCS 3586, Springer-Verlag (2005)
13. 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)
14. 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)
15. Ekman, T., Hedin, G.: The Jastadd extensible Java compiler. In: Proc. of ACM OOPSLA. pp. 1–18. ACM (2007)
16. Ernst, E.: Family polymorphism. In: ECOOP 2001 — Object-Oriented Programming. pp. 303–326. LNCS 2072, Springer-Verlag (2001)

17. 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)
18. 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)
19. Fowler, M.: Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html (2004)
20. Griswold, W.G., et al.: Modular software design with crosscutting interfaces. IEEE Software 23(1), 51–60 (2006)
21. Hansson, D.H., et al.: Ruby on Rails. http://rubyonrails.org (2003)
22. Haupt, M., Schippers, H.: A machine model for aspect-oriented programming. In: ECOOP 2007 – Object-Oriented Programming. LNCS, vol. 4609, pp. 501–524 (2007)
23. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (2008)
24. Ichisugi, Y., Tanaka, A.: Difference-based modules: A class-independent module mechanism. In: ECOOP 2002 – Object-Oriented Programming. pp. 62–88. LNCS 2374 (2002)
25. 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)
26. Johnson, R.: Expert One-on-One J2EE Design and Development. Wrox (2002)
27. 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)
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. Malabarba, S., et al.: Runtime support for type-safe dynamic Java classes. In: ECOOP 2000. pp. 337–361. LNCS 1850, Springer-Verlag (2000)
30. 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)
31. Millstein, T.: Practical predicate dispatch. In: Proc. of ACM OOPSLA. pp. 345–364. ACM (2004)
32. 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)
33. Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: Proc. of ACM OOPSLA. pp. 99–115 (2004)
34. 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)
35. 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)
36. Parnas, D.L.: Information distributions aspects of design methodology. In: Proc. of IFIP Congress '71. pp. 26–30 (1971)
37. Qi, X., Myers, A.C.: Sharing classes between families. In: Proc. of Conf. on Programming Language Design and Implementation. pp. 281–292 (2009)
38. 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)

39. 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)
40. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. Theory and Practice of Object Systems 2(3), 161–178 (1996)
41. Steimann, F.: The paradoxical success of aspect-oriented programming. ACM SIGPLAN Notices 41(10), 481–497 (2006)
42. 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)
43. The Eclipse Foundation: Eclipse IDE. http://www.eclipse.org (2001)
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)

A Proof of Theorems 1 and 2

Lemma 2 (Weakening). *If* $\mathtt{L}; \Gamma \vdash \mathtt{e} : \mathtt{C}$*, then* $\mathtt{L}; \Gamma, \mathtt{x}\colon \mathtt{D} \vdash \mathtt{e} : \mathtt{C}$.

Proof. By induction on $\mathtt{L}; \Gamma \vdash \mathtt{e} : \mathtt{C}$. □

Lemma 3. *If* $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ *and* $\mathtt{D} <: \mathtt{C}$*, then* $\mathit{fields}(\mathtt{D}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}, \overline{\mathtt{D}}\ \overline{\mathtt{g}}$ *for some* $\overline{\mathtt{D}}\ \overline{\mathtt{g}}$.

Proof. By induction on $\mathtt{D} <: \mathtt{C}$. □

Lemma 4. *If* $\mathit{mtype}(\mathtt{m}, \mathtt{L}_1, (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2), \mathtt{L}_2) = \overline{\mathtt{C}} \to \mathtt{C}$ *and* $\overline{\mathtt{R}} \supseteq (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2)$*, then* $\mathit{mtype}(\mathtt{m}, \mathtt{L}_1, \overline{\mathtt{R}}, \mathtt{L}_2) = \overline{\mathtt{C}} \to \mathtt{C}$.

Proof. By induction on the derivation of $\mathit{mtype}(\mathtt{m}, \mathtt{L}_1, (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2), \mathtt{L}_2) = \overline{\mathtt{C}} \to \mathtt{C}$. The only interesting case is $\mathtt{class\ L}_1\ \cdots\ \mathtt{\{\ [}\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;]}\ \overline{\mathtt{M}}\ \mathtt{\}}$ and $\mathtt{m\ [within\ L}_2\mathtt{]} \notin \overline{\mathtt{M}}$. Let ${\mathtt{L}_1}' = \mathit{super}(\mathtt{L}_1, (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2))$. By the induction hypothesis, we have

$$\mathit{mtype}(\mathtt{m}, {\mathtt{L}_1}', \overline{\mathtt{R}}, \mathtt{L}_2) = \overline{\mathtt{C}} \to \mathtt{C}$$
$$\mathit{mtype}(\mathtt{m}, {\mathtt{L}_1}', \mathit{dom}(RT), \mathtt{L}_2) = \overline{\mathtt{C}} \to \mathtt{C}.$$

If $\mathit{super}(\mathtt{L}_1, \overline{\mathtt{R}}) = {\mathtt{L}_1}'$, then, by MT-Super, $\mathit{mtype}(\mathtt{m}, \mathtt{L}_1, \overline{\mathtt{R}}, \mathtt{L}_2) = \overline{\mathtt{C}} \to \mathtt{C}$. Otherwise, if $\mathit{super}(\mathtt{L}_1, \overline{\mathtt{R}}) \neq {\mathtt{L}_1}'$, then there exists a sequence of class names ${\mathtt{L}_2}', \ldots, {\mathtt{L}_n}'$ such that ${\mathtt{L}'}_{i-1} = \mathit{super}({\mathtt{L}_i}', \overline{\mathtt{R}})$ and ${\mathtt{L}_n}' = \mathit{super}(\mathtt{L}_1, \overline{\mathtt{R}})$. By T-Method and T-MethodR, if $\mathtt{class\ L}_i{}'\ \cdots \mathtt{\{} \cdots \overline{\mathtt{M}}\mathtt{\}}$ and $\mathtt{D\ m(}\overline{\mathtt{D}}\ \overline{\mathtt{x}}\mathtt{)\{} \cdots \mathtt{\}\ [within\ L}_2\mathtt{]} \in \overline{\mathtt{M}}$, then $\overline{\mathtt{D}} = \overline{\mathtt{C}}$ and $\mathtt{D} = \mathtt{C}$ for any i. It follows that $\mathit{mtype}(\mathtt{m}, {\mathtt{L}_n}', \overline{\mathtt{R}}, \mathtt{L}_2) = \overline{\mathtt{C}} \to \mathtt{C}$ and MT-Super finishes the proof. □

Lemma 5. $\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{C}, (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2)), (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2), \mathtt{L}) = \mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{C}, (\overline{\mathtt{R}}_1, \mathtt{R}, \overline{\mathtt{R}}_2)), (\overline{\mathtt{R}}_1, \mathtt{R}, \overline{\mathtt{R}}_2), \mathtt{L})$.

Proof. The case where $\mathit{lowest}(\mathtt{C}, (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2)) = \mathit{lowest}(\mathtt{C}, (\overline{\mathtt{R}}_1, \mathtt{R}, \overline{\mathtt{R}}_2))$ follows from Lemma 4. Otherwise, it must be the case that $\mathit{lowest}(\mathtt{C}, (\overline{\mathtt{R}}_1, \mathtt{R}, \overline{\mathtt{R}}_2)) = \mathtt{R}$ and $\mathit{super}(\mathtt{R}, (\overline{\mathtt{R}}_1, \mathtt{R}, \overline{\mathtt{R}}_2)) = \mathit{lowest}(\mathtt{C}, (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2))$. Let $\mathtt{class\ R}\ \cdots\ \mathtt{\{}\ \overline{\mathtt{M}}\ \mathtt{\}}$. If $\mathtt{m\ [within\ L}\mathtt{]} \notin \overline{\mathtt{M}}$, then use MT-Super. Otherwise, $\mathtt{C\ m(}\overline{\mathtt{C}}\ \mathtt{x)\{} \cdots \mathtt{\}\ [within\ L}\mathtt{]} \in \overline{\mathtt{M}}$ and, by Lemma 4, we have to show that

$$\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{C}, (\overline{\mathtt{R}}_1, \overline{\mathtt{R}}_2)), \mathit{dom}(RT), \mathtt{L}) = \overline{\mathtt{C}} \to \mathtt{C}.$$

which follows from T-MethodR.

Lemma 6. *If* $\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{C}, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{C}} \to \mathtt{C}_0$ *and* $\mathtt{D} <: \mathtt{C}$*, then* $\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{D}, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{C}} \to \mathtt{C}_0$.

Proof. By Lemma 5, it suffices to show the case where $\overline{\mathtt{R}} = \mathit{dom}(RT)$. The proof proceeds by induction on $\mathtt{D} <: \mathtt{C}$. The only interesting case is when $\mathtt{D\ ext\ C}$; in that case, we have a sequence $\mathtt{L}_1, \ldots, \mathtt{L}_n$ of class names, where $\mathtt{L}_1 = \mathit{lowest}(\mathtt{D})$, $\mathtt{L}_{i+1} = \mathit{super}(\mathtt{L}_i)$, and $\mathtt{L}_n = \mathit{lowest}(\mathtt{C})$. By T-Method and T-MethodR, it is easy to show that $\mathit{mtype}(\mathtt{m}, \mathtt{L}_i, \mathit{dom}(RT), \mathtt{L}) = \mathit{mtype}(\mathtt{m}, \mathtt{L}_{i+1}, \mathit{dom}(RT), \mathtt{L})$ for any i. Induction on n finishes the case. □

Lemma 7 (Values are Location-Independent). *If* $\mathtt{L}; \Gamma \vdash \mathtt{v} : \mathtt{C}$*, then, for any* $\mathtt{L}'$*,* $\mathtt{L}'; \Gamma \vdash \mathtt{v} : \mathtt{C}$*.*

Proof. Easy induction on $\mathtt{L}; \Gamma \vdash \mathtt{v} : \mathtt{C}$. □

Lemma 8 (Substitution Preserves Typing). *If* $\mathtt{L}; \Gamma, \overline{\mathtt{x}}{:}\,\overline{\mathtt{C}} \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{L}; \Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}}$ *and* $\overline{\mathtt{D}} <: \overline{\mathtt{C}}$*, then there exists some* $\mathtt{D}$ *such that* $\mathtt{L}; \Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{D}$ *and* $\mathtt{D} <: \mathtt{C}$*.*

Proof. By induction on the derivation of $\mathtt{L}; \Gamma, \overline{\mathtt{x}}{:}\,\overline{\mathtt{C}} \vdash \mathtt{e} : \mathtt{C}$ with case analysis on the last rule used.

- Case T-Var is easy.
- Case T-Field. Then, we have $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$ and $\mathtt{L}; \Gamma, \overline{\mathtt{x}}{:}\,\overline{\mathtt{C}} \vdash \mathtt{e}_0 : \mathtt{C}_0$ and $\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$ and $\mathtt{C} = \mathtt{D}_i$ for some $\mathtt{e}_0$, $\mathtt{C}_0$, $\overline{\mathtt{D}}$, and $\overline{\mathtt{f}}$. By the induction hypothesis, for some $\mathtt{D}_0$, $\mathtt{L}; \Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{D}_0$ and $\mathtt{D}_0 <: \mathtt{C}_0$. By Lemma 3 and T-Field, we have $\mathtt{L}; \Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}](\mathtt{e}_0.\mathtt{f}_i) : \mathtt{C}$.
- Case T-Invk. Then, we have $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$ and $\mathtt{L}; \Gamma, \overline{\mathtt{x}}{:}\,\overline{\mathtt{C}} \vdash \mathtt{e}_0 : \mathtt{C}_0$ and $\mathtt{L}; \Gamma, \overline{\mathtt{x}}{:}\,\overline{\mathtt{C}} \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}}$ and $\mathtt{L}$ `using` $\overline{\mathtt{R}}$ and $\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{C}_0, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{E}} \to \mathtt{C}$ and $\overline{\mathtt{D}} <: \overline{\mathtt{E}}$ for some $\mathtt{e}_0$, $\overline{\mathtt{e}}$, $\mathtt{C}_0$, $\overline{\mathtt{D}}$, $\overline{\mathtt{R}}$, and $\overline{\mathtt{E}}$. By the induction hypothesis, there exist some $\mathtt{D}_0$ and $\overline{\mathtt{E}}'$ such that $\mathtt{L}; \Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{D}_0$ and $\mathtt{L}; \Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} : \overline{\mathtt{E}}'$ and $\mathtt{D}_0 <: \mathtt{C}_0$ and $\overline{\mathtt{E}}' <: \overline{\mathtt{D}}$. By Lemma 6, $\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{D}_0, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{E}} \to \mathtt{C}$ and, by S-Trans, $\overline{\mathtt{E}}' <: \overline{\mathtt{E}}$. The rule T-Invk finishes the case.
- Case T-New is easy.
- Case T-In is also easy by Lemma 7. □

Lemma 9. *If* $\mathit{mbody}(\mathtt{m}, \mathtt{L}_1, \mathtt{L}_2) = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{L}_3$ *and* $\mathit{mtype}(\mathtt{m}, \mathtt{L}_1, \mathit{dom}(RT), \mathtt{L}_2) = \overline{\mathtt{D}} \to \mathtt{D}_0$*, then there exists* $\mathtt{E}_0$ *such that* $\mathtt{L}_3; \overline{\mathtt{x}}{:}\,\overline{\mathtt{D}}, \mathtt{this}{:}\,\mathit{thistype}(\mathtt{L}_3) \vdash \mathtt{e}_0 : \mathtt{E}_0$ *and* $\mathtt{E}_0 <: \mathtt{D}_0$ *and* $\mathit{thistype}(\mathtt{L}_1) <: \mathit{thistype}(\mathtt{L}_3)$*.*

Proof. Easy induction on $\mathit{mbody}(\mathtt{m}, \mathtt{L}_1, \mathtt{L}_2) = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{L}_3$. □

Proof of Theorem 1. By induction on the derivation of $\mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ with case analysis on the last rule used.

- Case R-Field. Then, $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}).\mathtt{f}_i$ and $\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ and $\mathtt{e}' = \mathtt{v}_i$ for some $\mathtt{C}_0$, $\overline{\mathtt{f}}$, $\overline{\mathtt{v}}$, $\overline{\mathtt{C}}$. By T-Field and T-New, we have $\mathtt{L}; \Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}}$ and $\overline{\mathtt{D}} <: \overline{\mathtt{C}}$ and $\mathtt{C} = \mathtt{C}_i$ for some $\overline{\mathtt{D}}$. In particular, $\mathtt{L}; \Gamma \vdash \mathtt{v}_i : \mathtt{D}_i$ finishes the case.
- Case R-Invk. Then, $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{w}})$ and $\mathit{mbody}(\mathtt{m}, \mathit{lowest}(\mathtt{C}_0), \mathtt{L}) = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{L}'$ and $\mathtt{e}' = ([\overline{\mathtt{w}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})/\mathtt{this}]\mathtt{e}_0)$ `in` $\mathtt{L}'$ for some $\mathtt{C}_0$, $\overline{\mathtt{v}}$, $\overline{\mathtt{w}}$, $\overline{\mathtt{x}}$, $\mathtt{e}_0$, and $\mathtt{L}'$. By T-Invk and T-New, we have $\mathtt{L}; \Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}) : \mathtt{C}_0$ and $\mathtt{L}; \Gamma \vdash \overline{\mathtt{w}} : \overline{\mathtt{E}}$ and $\mathtt{L}$ `using` $\overline{\mathtt{R}}$ and $\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{C}_0, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{C}$ and $\overline{\mathtt{E}} <: \overline{\mathtt{D}}$ for some $\overline{\mathtt{E}}$, $\overline{\mathtt{R}}$, and $\overline{\mathtt{D}}$. By Lemmas 5 and 9, there exists $\mathtt{E}_0$ such that $\mathtt{L}'; \overline{\mathtt{x}}{:}\,\overline{\mathtt{D}}, \mathtt{this}{:}\,\mathit{thistype}(\mathtt{L}') \vdash \mathtt{e}_0 : \mathtt{E}_0$ and $\mathtt{E}_0 <: \mathtt{C}$ and $\mathit{thistype}(\mathit{lowest}(\mathtt{C}_0, \overline{\mathtt{R}})) = \mathtt{C}_0 <: \mathit{thistype}(\mathtt{L}')$. Then, by Lemmas 2, 7 and 8, there exists $\mathtt{E}_0{}'$ such that $\mathtt{L}'; \Gamma \vdash [\overline{\mathtt{w}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})/\mathtt{this}]\mathtt{e}_0 : \mathtt{E}_0{}'$ and $\mathtt{E}_0{}' <: \mathtt{E}_0$. By T-In, $\mathtt{L}; \Gamma \vdash ([\overline{\mathtt{w}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})/\mathtt{this}]\mathtt{e}_0)$ `in` $\mathtt{L}' : \mathtt{E}_0{}'$ and, by S-Trans, $\mathtt{E}_0{}' <: \mathtt{C}$, finishing the case.
- Case R-Return follows from Lemma 7.

– Case RC-Field follows from Lemma 3.
– Case RC-Invk-Recv follows from Lemma 6.
– Other cases are easy. □

Now we prove progress, which requires the following lemma:

Lemma 10. *If* $mtype(\mathtt{m}, \mathtt{L}_1, dom(RT), \mathtt{L}_2) = \overline{\mathtt{D}} \to \mathtt{D}_0$*, then* $mbody(\mathtt{m}, \mathtt{L}_1, \mathtt{L}_2) = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{L}_3$ *for some* $\overline{\mathtt{x}}$*, whose length is the same as that of* $\overline{\mathtt{D}}$*, and* $\mathtt{e}_0$ *and* $\mathtt{L}_3$*.*

Proof. Easy induction on the derivation of $mtype(\mathtt{m}, \mathtt{L}_1, dom(RT), \mathtt{L}_2) = \overline{\mathtt{D}} \to \mathtt{D}_0$. □

Proof of Theorem 2. By induction on the structure of $\mathtt{e}$.

– Case $\mathtt{e} = \mathtt{x}$ cannot happen since the type environment is empty.
– Case $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$. By T-Field, $\mathtt{L}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ and $fields(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ and $\mathtt{C}_i = \mathtt{C}$ for some $\mathtt{C}_0$ and $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$. If $\mathtt{e}_0$ is a value, then it must be the case that $\mathtt{e}_0 = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})$ and $\mathtt{L}; \Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}}$ and $\overline{\mathtt{D}} <: \overline{\mathtt{C}}$, so $\mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{v}_i$ by R-Field. Otherwise, by the induction hypothesis, $\mathtt{L} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0'$ for some $\mathtt{e}_0'$ and, by RC-Field, $\mathtt{L} \vdash \mathtt{e}_0.\mathtt{f}_i \longrightarrow \mathtt{e}_0'.\mathtt{f}_i$.
– Case $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$. By T-Invk, $\mathtt{L}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ and $\mathtt{L}; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$ and $\mathtt{L}$ `using` $\overline{\mathtt{R}}$ and $mtype(\mathtt{m}, lowest(\mathtt{C}_0, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{C}$ and $\overline{\mathtt{C}} <: \overline{\mathtt{D}}$ for some $\mathtt{C}_0$, $\overline{\mathtt{C}}$, $\overline{\mathtt{R}}$, and $\overline{\mathtt{D}}$. If one of $\mathtt{e}_0$, $\overline{\mathtt{e}}$ is not a value, then by the induction hypothesis, there exists an expression $\mathtt{e}_i'$ such that $\mathtt{L} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i'$ and by RC-Invk-Recv or RC-Invk-Arg, $\mathtt{L} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ for some $\mathtt{e}'$. Otherwise, all of $\mathtt{e}_0$, $\overline{\mathtt{e}}$ are values. In particular, it must be the case that $\mathtt{e}_0 = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})$ by T-New. By Lemmas 5 and 10, we have $mbody(\mathtt{m}, lowest(\mathtt{C}_0), \mathtt{L}) = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{L}'$ for some $\overline{\mathtt{x}}$, whose length is the same as that of $\overline{\mathtt{D}}$ (and $\overline{\mathtt{e}}$), and $\mathtt{e}_0$ and $\mathtt{L}'$. By R-Invk, $\mathtt{L} \vdash \mathtt{e} \longrightarrow ([\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})/\mathtt{this}]\mathtt{e}_0)$ `in` $\mathtt{L}'$.
– Other cases are easy. □

B Proof of Theorem 4

Lemma 11. *If* $(CT, RT) \Longrightarrow CT'$*, then* $fields(\mathtt{C}) = fields(\mathtt{L})$ *under* CT' *for any* $\mathtt{C} \in dom(CT)$ *and* $\mathtt{L} = lowest(\mathtt{C})$ *(under* (CT, RT)*).*

Proof. Easy since no revising classes have fields. □

Lemma 12. *Suppose* $(CT, RT) \Longrightarrow CT'$*. Then* $mtype(\mathtt{m}, \mathtt{L}_0, dom(RT), \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{C}$ *under* (CT, RT) *if and only if* $mtype(\mathtt{m}, \mathtt{L}_0, \bullet, \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{C}$ *under* CT'*.*

Proof. By induction on the derivation of $mtype(\mathtt{m}, \mathtt{L}_0, dom(RT), \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{C}$. The other direction is similar. □

Lemma 13. *Suppose* $(CT, RT) \Longrightarrow CT'$ *and* $\mathtt{L} \in within(\mathtt{m}, thistype(\mathtt{L}_0))$*. Then,* $mtype(\mathtt{m}, \mathtt{L}_0, dom(RT), \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{D}$ *under* (CT, RT) *if and only if* $mtype(\mathtt{m}_\mathtt{L}, \mathtt{L}_0) = \overline{\mathtt{D}} \to \mathtt{D}$ *under* CT'*.*

Proof. Similar to Lemma 12. □

Proof of Lemma 1. By induction on $\mathtt{L}; \Gamma \vdash \mathtt{e} : \mathtt{C}$ with case analysis on the last typing rule used.

- Case T-Var. Trivial.
- Case T-Field. We have $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$ and $\mathtt{L}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ and $\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ and $\mathtt{C} = \mathtt{C}_i$. Then, it must be the case that $\mathtt{L}; \Gamma \vdash \mathtt{e}_0 \Longrightarrow \mathtt{e}_0'$ and $\mathtt{e}' = \mathtt{e}_0'.\mathtt{f}_i$. By the induction hypothesis, $\mathtt{L}; \Gamma \vdash \mathtt{e}_0' : \mathtt{L}_0$ and $\mathtt{L}_0$ is either $\mathtt{C}_0$ or $\mathit{lowest}(\mathtt{C}_0)$. For the latter case, by Lemma 11, $\mathit{fields}(\mathit{lowest}(\mathtt{C}_0)) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. So, in either case, $\mathtt{L}; \Gamma \vdash \mathtt{e}' : \mathtt{C}_i$.
- Case T-Invk. Then, we have $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$ and $\mathtt{L}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ and $\mathtt{L}; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$ and $\mathtt{L}$ `using` $\overline{\mathtt{R}}$ and $\mathit{mtype}(\mathtt{m}, \mathit{lowest}(\mathtt{C}_0, \overline{\mathtt{R}}), \overline{\mathtt{R}}, \mathtt{L}) = \overline{\mathtt{D}} \to \mathtt{C}$ and $\overline{\mathtt{C}} <: \overline{\mathtt{D}}$. It must be the case that $\mathtt{e}' = ((\mathtt{L}')\mathtt{e}_0').\mathtt{m}'(\overline{\mathtt{e}'})$ and $\mathtt{L}; \Gamma \vdash \mathtt{e}_0 \Longrightarrow \mathtt{e}_0'$ and $\mathtt{L}; \Gamma \vdash \overline{\mathtt{e}} \Longrightarrow \overline{\mathtt{e}'}$ and $\mathtt{L}' = \mathit{lowest}(\mathtt{C}_0, \overline{\mathtt{R}})$ and $\mathtt{m}'$ is either $\mathtt{m}_{\mathtt{L}}$ (if $\mathtt{L} \in \mathit{within}_{sub}(\mathtt{m}, \mathtt{C}_0)$), or $\mathtt{m}$ (otherwise). We show the former case. By the induction hypothesis, $\mathtt{L}; \Gamma \vdash \mathtt{e}_0' : \mathtt{E}_0$ and $\mathtt{L}; \Gamma \vdash \overline{\mathtt{e}'} : \overline{\mathtt{E}}$ and each $\mathtt{E}_i$ is either $\mathtt{C}_i$ or $\mathit{lowest}(\mathtt{C}_i)$. We also have $\mathtt{L} \in \mathit{within}(\mathtt{m}, \mathtt{C}_0)$ since it is easy to show that $\mathit{within}_{sub}(\mathtt{m}, \mathtt{C}_0) \subseteq \mathit{within}(\mathtt{m}, \mathtt{C}_0)$. By T-UCast or T-DCast, $\mathtt{L}; \Gamma \vdash (\mathtt{L}')\mathtt{e}_0' : \mathtt{L}'$. By $\mathit{lowest}(\mathtt{L}', \overline{\mathtt{R}}) = \mathtt{L}'$ and $\mathit{thistype}(\mathtt{L}') = \mathtt{C}_0$ (under CT') and Lemmas 4 and 13, $\mathit{mtype}(\mathtt{m}, \mathtt{L}', \overline{\mathtt{R}}, \mathtt{L}) = \mathit{mtype}(\mathtt{m}, \mathtt{L}', \mathit{dom}(RT), \mathtt{L}) = \mathit{mtype}(\mathtt{m}_{\mathtt{L}}, \mathtt{L}', \bullet, \mathtt{L})$. By S-Trans, $\overline{\mathtt{E}} <: \overline{\mathtt{D}}$. So, by T-Invk, we have $\mathtt{L}; \Gamma \vdash \mathtt{e}' : \mathtt{C}$.
  The other case is similar.
- Case T-New is easy by Lemma 11. □

Lemma 14. *If* $(CT, RT) \Longrightarrow CT'$ *and* $\mathtt{M}$ `OK IN` $\mathtt{L}$ *under* (CT, RT) *and* $\mathtt{L} \vdash \mathtt{M} \Longrightarrow \overline{\mathtt{M}}$*, then* $\overline{\mathtt{M}}$ `OK IN` $\mathtt{L}$ *(under* CT'*).*

Proof. We first show the case $\mathtt{L} = \mathtt{C}$ for some $\mathtt{C}$. By Tr-Method, $\mathtt{M} =$ `B m(`$\overline{\mathtt{B}}$ $\overline{\mathtt{x}}$`){ return e; }` and $\mathtt{C}; \overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} \Longrightarrow \mathtt{e}'$ and

$$\overline{\mathtt{M}} = \begin{array}{l} \mathtt{B\ m(}\overline{\mathtt{B}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e';\ \}} \\ \mathtt{B\ m}_{\mathtt{L}_1}\mathtt{(}\overline{\mathtt{B}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e';\ \}} \\ \qquad\qquad \vdots \\ \mathtt{B\ m}_{\mathtt{L}_n}\mathtt{(}\overline{\mathtt{B}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e';\ \}} \end{array}$$

where $\mathit{within}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{L}}$. By T-Method,

$$\mathtt{C}; \overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} : \mathtt{C}_0 \qquad \mathtt{C}_0 <: \mathtt{B}.$$

By Lemma 1, we have

$$\mathtt{C}; \overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}' : \mathtt{D}_0$$

for some $\mathtt{D}_0$ such that $\mathtt{D}_0 <: \mathtt{B}$. Now, suppose $\mathit{mtype}(\mathtt{m}_{\mathtt{L}_i}, \mathit{super}(\mathtt{C})) = \overline{\mathtt{D}} \to \mathtt{D}$. Then, by Lemma 13, $\mathit{mtype}(\mathtt{m}, \mathit{super}(\mathtt{C}), \mathit{dom}(RT), \mathtt{L}_i) = \overline{\mathtt{D}} \to \mathtt{D}$. Then, by T-Method, $\overline{\mathtt{D}} = \overline{\mathtt{B}}$ and $\mathtt{D} = \mathtt{B}$. So, by T-Method,

$$\mathtt{B\ m}_{\mathtt{L}_i}\mathtt{(}\overline{\mathtt{B}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e';\ \}\ OK\ IN\ C}.$$

`B m(`$\overline{\mathtt{B}}$ $\overline{\mathtt{x}}$`){ return e'; } OK IN C` is similar.

Lemma 15. *If* $(CT, RT) \Longrightarrow CT'$ *and* `CL OK` *under* (CT, RT) *and* $\vdash$ `CL` $\Longrightarrow$ `CL′`*, then* `CL′ OK` *(under* CT'*).*

Proof. It must be the case that `CL` = `class L {extends,revises} D using` $\overline{\texttt{R}}$ `{` $\overline{\texttt{C}}$ $\overline{\texttt{f}}$`;` $\overline{\texttt{M}}$ `}` and *super*(`L`) = `L′` and `L` $\vdash$ $\overline{\texttt{M}} \Longrightarrow \overline{\texttt{M}}'$ and `CL′` = `class L extends L′ {` $\overline{\texttt{C}}$ $\overline{\texttt{f}}$`;` $\overline{\texttt{M}}$ `}`. By Lemma 14, $\overline{\texttt{M}}'$ `OK IN L`. □

Proof of Theorem 4. It is easy to show the existence of CT'. Then, Lemma 15 finishes the proof. □