Selective Open Recursion: A Solution to the Fragile Base Class Problem

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

ABSTRACT

Current object-oriented languages do not fully protect the implementation details of a class from its subclasses, making it difficult to evolve that implementation without breaking subclass code. Previous solutions to the so-called fragile base class problem *specify* those implementation dependencies, but do not *hide* implementation details in a way that allows effective software evolution.

In this paper, we show that the fragile base class problem arises because current object-oriented languages dispatch methods using *open recursion* semantics even when these semantics are not needed or wanted. Our solution is to make explicit the methods to which open recursion should apply.

We propose to change the semantics of object-oriented dispatch, such that all calls to "open" methods are dispatched dynamically as usual, but calls to "non-open" methods are dispatched *statically* if called on the current object `this`, but *dynamically* if called on any other object. By specifying a method as open, a developer is promising that future versions of the class will make internal calls to that method in exactly the same way as the current implementation. Because internal calls to non-open methods are dispatched statically, developers can change the way these methods are called without affecting subclasses.

We have implemented Selective Open Recursion as an extension to Java. To verify the correctness of our design, we formalize it as an extension to Featherweight Java. We then prove a variant of Reynolds' abstraction theorem for our system. This modularity property shows that a class can be changed without affecting its subclasses, as long as the method implementations have the same functional behavior, and as long as the two implementations invoke open methods under the same conditions. Thus, developers using our system can evolve a class with confidence that their changes will not affect subclasses.

1. Introduction

In his seminal paper, Parnas laid out the classic theory of information hiding: developers should break a system into modules in order to hide information that is likely to change [13]. Thus if change is anticipated with reasonable accuracy, the system can be evolved with local rather than global system modifications, easing many software maintenance tasks. Furthermore, the correctness of each module can be verified in isolation from other modules, allowing developers to work independently on different sub-problems.

```
public class CountingSet extends Set {
  private int count;

  public void add(Object o) {
    super.add(o);
    count++;
  }
  public void addAll(Collection c) {
    super.addAll(c);
    count += c.size();
  }
  public int size() {
    return count;
  }
}
```

Figure 1: The correctness of the `CountingSet` class depends on the independence of `add` and `addAll` in the implementation of `Set`. If the implementation is changed so that `addAll` uses `add`, then `count` will be incremented twice for each element added.

Unfortunately, developers do not always respect the information hiding boundaries of modules–it is often tempting to reach across the boundary for temporary convenience, while causing more serious long-term evolution problems. Thus, encapsulation mechanisms such as Java's packages and public/private data members were developed to give programmers compiler support for enforcing information hiding boundaries.

While the encapsulation mechanisms provided by Java and other languages can help to enforce information hiding boundaries between an object and its clients, enforcing information hiding between a class and its subclasses is more challenging. The `private` modifier can be used to hide some method and fields from subclasses. However, inheritance creates a tight coupling between a class and its subclasses, making it difficult to hide information about the implementation of `public` and `protected` methods in the superclass.

1.1 The Fragile Base Class Problem

The breakdown of information hiding in the presence of inheritance causes the *Fragile Base Class Problem*, one of the most significant challenges faced by designers of object-

oriented libraries. Figure 1 shows an example of the fragile base class problem, taken from the literature [15, 11, 4].

In the example, the `Set` class has been extended with an optimization that keeps track of the current size of the set in an additional variable. Whenever a new element or collection of elements is added to the set, the variable is updated appropriately.

Unfortunately, the implementation of `CountingSet` makes assumptions about the implementation details of `Set`–in particular, it assumes that `Set` does not implement `addAll` in terms of `add`. This coupling means that the implementation of `Set` cannot be changed without breaking its subclasses. For example, if `Set` was changed so that the `addAll` method calls `add` for each member of the collection in the argument, the `count` variable will be updated not only during the call to `addAll`, but also for each individual `add` operation–and thus it will end up being incremented twice for each element in the collection.

The core of the problem is the *open recursion* provided by inheritance–that is, the fact that method calls on the current object `this` are dynamically dispatched, and can therefore be intercepted and observed by subclass code. Open recursion is useful for many object-oriented programming idioms–for example, the template method design pattern [8] uses open recursion to invoke customized code provided by a subclass. However, sometimes making a self-call to the current object is just an implementation convenience, not a semantic requirement. The whole point of encapsulation is to ensure that subclasses do not depend on such implementation details, so that a class and its subclasses can be evolved independently. Thus inheritance breaks encapsulation when implementation-specific self-calls are made.

1.2 Previous Approaches

Although the example in the figure above is simplistic, the fragile base class problem presents a severe challenge to object-oriented library designers. Joshua Bloch, one of the principal designers of the Java standard libraries, considers the problem so significant that he recommends that library designers make their classes `final` so they cannot be extended by inheritance at all [4]. Bloch, Szyperski, and others suggest using delegation in place of inheritance [4, 16], but as Szyperski notes, not all uses of inheritance can be replaced by delegation because open recursion is sometimes needed [16].

A number of researchers have proposed to address the fragile base class problem by documenting calling dependencies between methods [10, 15, 14]. In these proposals, the author of a class documents which methods call which other methods, and subclasses can rely on these dependencies. Unfortunately, these proposals solve the fragile base class problem not by hiding implementation details, but rather by exposing them. Since the calling patterns of a class are documented in the subclassing interface–and since subclasses may depend on them–making significant changes to the implementation of the class become impossible.

1.3 Contribution

The contribution of this paper is Selective Open Recursion, a new approach that provides the benefits of inheritance and open recursion where they are needed, but allows programmers to effectively hide many details of the way a class is implemented. In our system, method calls on the current object `this` are dispatched statically by default, meaning that subclasses cannot intercept internal calls and thus cannot become dependent on those implementation details. External calls to the methods of an object–i.e., any method call not explicitly invoked on `this`–are dynamically dispatched as usual.

If an engineer needs open recursion, she can declare a method "open," in which case self-calls to that method are dispatched dynamically. By declaring a method "open," the author of a class is promising that any changes to the class will preserve the ways in which that method is called.

We have implemented our proposal in a Java compiler. We are also in the process of implementing an analysis that can annotate an existing Java program with the minimal set of "open" declarations that are necessary so that the program has the same semantics in our system. This analysis could be used to automatically and safely translate existing Java programs into a system supporting Selective Open Recursion.

To validate our proposal in theory, we model it as an extension to Featherweight Java. We prove a variant of Reynolds' abstraction theorem in our system: if two implementations of a class are bisimilar with respect to the open methods in that class, then no client or subclass can distinguish these implementations. The abstraction theorem is the core property underlying modularity and information hiding, in that it proves that the implementation of class can be changed without affecting clients. To the best of our knowledge, our work is the first to prove this abstraction property for a language with inheritance.

1.4 Outline

In the next section we provide more details on our Selective Open Recursion proposal. Section 3 describes our implementation of Selective Open Recursion. Section 4 describes the design of an analysis that can be used to safely and automatically transform existing Java code into our system. Section 5 models Selective Open Recursion in Featherweight Java, and proves an abstraction result. Section 6 describes related work in more detail, and Section 7 concludes.

2. Selective Open Recursion

We argue that the issue underlying the fragile base class problem is that current languages do not allow programmers to adequately express the intent of various methods in a class. There is an important distinction between methods used for communication between a class and its clients, vs. methods used for communication between a class and its subclasses.

Some methods are specifically intended as callbacks or extension points for subclasses. These methods are invoked recursively by a class so that its subclasses can provide customized behavior. Examples of callback methods include methods denoting events in a user interface, as well as abstract "hook" methods in the template method design pattern [8]. Because callback methods are intended to be invoked whenever some semantic event occurs, any changes to the base class must maintain the invariant that the method is always invoked in a consistent way. The fragile base class problem cannot arise in this setting, because the correct semantics of the method already require that the method be called in a consistent way.

In contrast, many accessor and mutator functions are primarily intended for use by clients. If the implementation of a class also uses these functions, it is typically as a conve-

```
public class Set {
  List elements;

  public void add(Object o) {
    if (!elements.contains(o))
      elements.add(o);
  }
  public void addAll(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext())
      add(i.next());
  }
}
```

Figure 2: In the first solution to the problem described in Figure 1, the developer decides not to mark either `add` or `addAll` as `open`. Thus, when `addAll` invokes `add`, the call is dispatched statically, so that `Set`'s implementation of `add` executes even if a subclass overrides the `add` method (Client calls to `add` are dispatched statically as usual). Thus, subclasses cannot tell if `addAll` was implemented in terms of `add` or not, allowing the maintainer of `Set` to change this decision.

```
public class Set {
  List elements;

  public void open add(Object o) {
    if (!elements.contains(o))
      elements.add(o);
  }
  public void addAll(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext())
      add(i.next());
  }
}
```

Figure 3: In the second solution to the problem described in Figure 1, the developer decides that the `add` method denotes a semantic event of interest to subclasses, and therefore marks `add` as `open`. By doing this, the developer is promising that any correct implementation of `Set` will call `add` once for each element added to the set. Therefore, a subclass interested in "add element" events can override the `add` method without overriding `addAll`.

nience, *not* because the class expects subclasses to override the function with customized behavior. The fragile base class problem occurs exactly when a "client-oriented" method is called recursively by a superclass, but is also overridden by a subclass. Because the recursive call to the method was never intended to be part of the subclassing interface, the maintainer of the base class should be able to evolve the class to use (or not use) such methods without affecting subclasses.

The key insight underlying Selective Open Recursion is that *the circumstances where the fragile base class problem may arise are exactly those circumstances where open recursion is not needed*. Subclasses do not need to intercept recursive calls to methods that were not intended as callbacks or extension points–they can always provide their behavior by overriding the external interface of a class. At most, intercepting recursive calls to "client-oriented" methods is only a minor convenience, and one that creates an undesirable coupling between subclass and superclasses.

We propose to add a new modifier, `open`, which allows developers to more fully declare their underlying design intent. An `open` method has open recursion semantics; it is treated as a callback for subclasses that will always be recursively invoked by the superclass whenever some conceptual event occurs. Ordinary methods–those without the `open` keyword–are not part of the subclassing interface. While external calls to ordinary methods are dynamically dispatched as usual, recursive calls where the receiver is `this` are dispatched statically. Because open recursion does not apply to methods that are not marked `open`, subclasses cannot depend on when they are invoked, and the fragile base class problem cannot occur.

In our proposal, there are two choices a designer can make to solve the problem described in Figure 1. In the first solution, shown in Figure 2, the designer of the `Set` class has decided that neither `add` and `addAll` are intended to act as subclass callbacks, and so neither method was annotated `open`. In this case, subclasses cannot tell whether `addAll` is implemented in terms of `add` or not, and so the fragile base class problem cannot arise. Even if `addAll` calls `add` on the current object `this` (which is implicit in the example), this call will be dispatched statically and so subclasses cannot intercept it. Note that calls to `add` from clients are dispatched dynamically as usual, so that an implementation of `CountingSet` can accurately track the element count simply by overriding both `add` and `addAll`.

In the second solution, shown in Figure 3, the designer of the `Set` class has decided that `add` represents a semantic event (adding an element to the set) that subclasses may be interested in reacting to. The designer therefore annotates `add` as `open`, documenting the promise that even if the implementation of `Set` changes, the `add` method will always be called once for each element added to the set. The implementor of `CountingSet` can keep track of the element count by overriding just the `add` function. Any changes to the `Set` class will not break the `CountingSet` code, because the implementor of `Set` has promised that any changes to `Set` will preserve the semantics of calls to `add`.

2.1 Using Selective Open Recursion

With any new language construct, it is important not only to describe the construct's meaning but also how to use it effectively. We offer tentative guidelines for the use of `open`, which can be refined as experience is gained with the construct.

We expect that `public` methods will generally not be `open`. The rationale for this guideline is that `public` methods are intended for use by clients, not by subclasses. In general, any internal use of these public methods is probably coincidental, and subclasses should not rely on these calls. There are exceptions–for example, the `add` method could be both `public` and `open`, depending on the designer's intent–but these idioms can also be expressed by having the `public` method invoke a `protected open` method. For example, instead of making the `add` method `open`, the developer could implement both `add` and `addAll` in terms of a `protected, open internalAdd` method that serves as the subclass extension point. Using this `protected` method

solution is potentially cleaner because it separates the client interface from the subclassing interface.

On the other hand, we expect that `protected` methods will either be `final` or `open`. Protected methods are usually called on the current object `this`, so overriding them is useful only in the presence of open recursion. Protected methods that are not intended to represent callbacks or extension points for subclasses should be marked as `final`.

Private methods in languages like Java are unaffected by our proposal; since they cannot be overridden, open recursion is not relevant.

2.2 An Alternative Proposal

The discussion above suggests an alternative proposal: instead of adding a new keyword to the programming language, simply use open recursion dispatch semantics for all (non-`final`) `protected` methods and treat all `public` methods as if they were non-open. This alternative has the advantage of simplicity; it takes advantage of common patterns of usage, does not add a new keyword to the language, and encourages programmers to cleanly separate the public client interface from the protected subclass interface.

However, there are two disadvantages to the alternative. If a public method also represents an event that subclasses may want to extend, the programmer will be forced to create an additional protected method for the subclass interface, creating a minor amount of code bloat. Furthermore, the proposal that makes `open` explicit is a more natural evolutionary path; existing Java code need only be annotated with `open` (perhaps with the analysis described in Section 4), whereas in the alternative proposal `public` methods that are conceptually `open` would have to be re-written as a pair of `public` and `protected` methods.

2.3 Applications to Current Languages

Our proposal extends languages like Java and C# in order to capture more information about how a class can be extended by subclasses. However, the idea of "open" methods can also be applied within existing languages, providing engineering guidelines for avoiding problematic uses of inheritance.

The discussion above suggests that developers should avoid calling `public` methods on the current object `this`. If a `public` method contains code that can be reused elsewhere in the class, the code should be encapsulated in a `protected` or `private` method, and the `public` method should call that internal method. This guideline was previously suggested by Ruby and Leavens [14], and appears to be common practice within the Java standard library in any case. For example, the `java.util.Vector` class in the JDK 1.4.2 internally calls a `protected` method, `ensureCapacityHelper`, to verify that the underlying array is large enough–even though the `public` method `ensureCapacity` could be used instead.

Protected methods should be `final` if they don't represent an explicit extension point for subclasses. The author of a library should carefully document under which circumstances non-`final` protected methods are called, so that subclasses can rely on the semantics.

2.4 A Rejected Alternative Design

Based on the insight that the fragile base class problem arises when open recursion is used unintentionally, there is a natural alternative design to be considered. In the discussion above, we chose to annotate methods as being `open` or not; an alternative is to annotate call sites as using dynamic or static dispatch. We rejected this alternative for two reasons. First, it is a poor match for the design intent, which associates a method–not a call site–with a callback or extension point. Second, because the design intent is typically associated with methods, it would be very surprising if different recursive calls to the same method were treated differently. By annotating the method rather than the call site, our proposal helps developers be consistent.

2.5 Family Polymorphism

The fragile base class problem can be generalized to sets of classes that are closely related. For example, if a `Graph` module defines classes for nodes and edges, it is likely that the node and edge class are closely related and will often be inherited together. Just as self-calls in an object-oriented setting can be mistakenly "captured" by subclasses, calls between node and edge superclasses might be mistakenly captured by node and edge subclasses.

This paper is primarily focused on the version of the problem that is restricted to a single subclass and superclass, partly because the right solution is more clear-cut in this setting. However, some languages provide first-class support for extending related classes together through mechanisms like Family Polymorphism [7]. In this setting, our proposal could potentially be generalized to distinguish between inter-object calls that should be dispatched dynamically and those that should be dispatched statically. Further work is needed to understand how to apply our proposal effectively in this setting.

2.6 Pure Methods

A central aspect of our approach is that a class must document the circumstances under which all of its open methods are called internally. As suggested by Ruby and Leavens [14], it is possible to relax this requirement for *pure* methods which have no (visible) side-effects. Since these methods have no effects, a class can change the way in which they are called without affecting subclasses. An auxiliary analysis or type system could be used to verify that pure methods have no effects, including state changes (other than caches), I/O operations, or non-termination.

3. Implementation

We have implemented Selective Open Recursion as an extension to the Barat Java compiler [5]. Our implementation strategy leaves `open` methods and `private` methods unchanged. For each non-open `public/protected` method in the source program, we generate another `protected final` method containing the implementation, and rewrite the original method to call the new method. We leave all calls to `open` methods unchanged, as well as all calls to methods with a receiver other than `this`. For every call to a non-open method that has `this` as the receiver, including implicit uses of `this`, we call the corresponding implementation method, thus simulating static dispatch.

Our implementation of Selective Open Recursion is available at `http://www.archjava.org/` as part of the open source ArchJava compiler.

4. Inference of Open Recursion

In order to ease a potential transition from standard Java or C# to a system with Selective Open Recursion, we are implementing an analysis that can automatically infer which methods must be annotated with `open` in order to preserve the original program's semantics. Of course, our system is identical to Java-like languages if every method is `open`, so the goal of the analysis is to introduce as few `open` annotations as possible. Extra `open` annotations are problematic because they create the possibility of using open recursion when it was not intended, thus triggering the fragile base class problem. In general, no analysis can do this perfectly, because the decision to make a method `open` is a design decision that may not be expressed explicitly in the source code. However, an analysis can provide a reasonable (and safe) default that can be refined manually later.

In order to gain precision, our analysis design assumes that whole-program information is available. A local version of the analysis could be defined, but it would have to assume that every method called on `this` is `open`, because otherwise some unknown subclass could rely on the open recursion semantics of Java-like languages. This assumption would be extremely conservative, so much so that it would be likely to obscure any potential benefits of Selective Open Recursion.

Our analysis design examines each `public` and `protected` method m of every class C. The program potentially relies on open recursion for calls to m whenever there is some method m' in a subclass C' that calls m on `this`, and some subclass C'' of C' overrides m, and that subclass either doesn't override m' or makes a super call to m'. The analysis conservatively checks this property, and determines that the method should be annotated `open` whenever the property holds.

5. Formalization

We would like to define the semantics of Selective Open Recursion precisely, and use formal techniques to prove that Selective Open Recursion eliminates the fragile base class problem. A standard technique, exemplified by Featherweight Java [9], is to formalize a core language that captures the key semantic issues while ignoring complicating language details. We have formalized Selective Open Recursion as Featherweight OpenJava (FOJ), a core language based on Featherweight Java (FJ).

Featherweight OpenJava makes a number of simplifications relative to the full Java language. As in FJ, the model omits interfaces, inner classes, and some statement and expression forms, since these constructs can be written in terms of more fundamental ones. These omissions make the formal system simple enough to permit effective reasoning, while still capturing the core of our Selective Open Recursion proposal.

In fact, Featherweight OpenJava is almost indistinguishable from the FJ Proposal–it adds only the `open` keyword and constructs for modeling static dispatch when non-open methods invoked on `this`. For completeness, we include all the relevant definitions and briefly discuss their semantics. The reader who is already familiar with Featherweight Java may wish to read only the paragraphs in the next few sections marked "New Constructs."

$$
\begin{array}{rcl}
CL & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \\
K & ::= & C(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\{\ \texttt{super}(\overline{g}); \texttt{this}.\overline{f} = \overline{f};\} \\
M & ::= & [\texttt{open}]\ C\ m(\overline{C}\ \overline{x})\ \{\ \texttt{return}\ e;\ \} \\
e & ::= & x \mid \texttt{new}\ C(\overline{e}) \\
 & \mid & e.f \mid (C)\ e \mid e.m(\overline{e}) \\
 & \mid & e.C{::}m(\overline{e}) \\
v & ::= & \texttt{new}\ C(\overline{v}) \\
\Gamma & ::= & x \mapsto C
\end{array}
$$

Figure 4: Featherweight OpenJava Syntax

5.1 Syntax

Figure 4 shows the syntax of FOJ. The metavariables C, D, and E range over class names; f and g range over field names; v ranges over values; e ranges over expressions; x ranges over variable names; and m ranges over method names. As a shorthand, an overbar is used to represent a sequence.

In FOJ, each class extends another class, possibly the predefined class `Object`. Each class defines a set of fields, constructors, and methods. Constructors just assign the class's fields to the constructor arguments, while the bodies of methods return a single expression. Expressions include object creation expressions, field reads, casts, and method calls. The predefined class `Object` has no fields or methods.

The result of computation is a value v, which is an object creation expression with values for all the object's fields. The set of variables x includes the distinguished variable `this` used to refer to the receiver of a method.

Types in FOJ are simply class names, and a typing environment Γ maps variables x to their types. We assume a fixed class table CT mapping classes to their definitions. A program, then, is a pair (CT, e) of a class table and an expression.

New Constructs. Featherweight OpenJava allows any method to be modified by the `open` keyword, indicating that open recursion should apply to calls to that method. OpenJava also includes a construct for static method calls, written $e.C{::}m(\overline{e})$, similar to the C++ syntax for super calls. The static method call form may not appear in the source text of the program. Instead, when a method is invoked on a class C, all calls to non-open methods on the current object `this` in the body of the method are re-written to use static rather than dynamic dispatch.

5.2 New Constructs: Expressiveness

We deliberately leave mutable state out of Featherweight OpenJava because we wish to prove Reynolds' abstraction theorem, a strong modularity property, for our system. Proving an abstraction theorem in the presence of arbitrary mutable state is an open problem, although Banerjee and Naumann have proved abstraction for restricted uses of state [3].

This choice might seem to compromise the goals of the paper. We have already observed that if all methods have no effects, the fragile base class problem cannot occur. It might ap-

```
class List extends Object {
  List add(Object o) {
    return new Cons(o,this);
  }

  /* first implementation */
  List addAll(List l) {
    if (l instanceof Nil) {
      return this;
    } else {
      Cons c = (Cons) l;
      return this.add(c.first)
                 .addAll(c.rest);
    }
  }

  /* second implementation */
  List addAll(List l) {
    if (l instanceof Nil) {
      return this;
    } else {
      Cons c = (Cons) l;
      return new Cons(c.first, this)
                 .addAll(c.rest);
    }
  }
}

class Nil extends List {
}

class Cons extends Nil {
  Object first;
  List rest;
}

class EffectfulCons extends Cons {
  List add(Object o) {
    this.add(o);
  }
}
```

Figure 5: A purely functional list class with two different implementations, along with a subclass `EffectfulCons` that can distinguish them in Java (but not in our proposal).

pear that Featherweight OpenJava is uninteresting, because FOJ does not model effects such as mutable state and I/O. However, FOJ does allow us to model non-termination as an effect, and non-termination is sufficient to exhibit the fragile base class problem in Java.

For example, Figure 5 shows a purely functional linked list class with two possible implementations for the method `addAll`. The implementations are identical except that the first calls `add` recursively on `this`, whereas the second inlines the body of `add` into `addAll`.

From the perspective of a client, these implementations have identical semantics, and so we would expect that they should be indistinguishable. However, a client can distinguish the two implementations (in Java) by defining subclass `EffectfulCons` that overrides the `add` method with

$$\begin{array}{lll} \mathit{Contexts} & \mathit{Ctx} ::= & \mathsf{new}\, C(v_1, \ldots, v_{i-1}, \Box, e_{i+1}, \ldots, e_n) \\ & | & \Box.f \mid \Box.[C::]m(\overline{e}) \mid (C)\,\Box \\ & | & v.[C::]m(v_1, \ldots, v_{i-1}, \Box, e_{i+1}, \ldots, e_n) \end{array}$$

$$\frac{\mathit{fields}(C) = \overline{C}\ \overline{f}}{\mathsf{new}\, C(\overline{v}).f_i \mapsto v_i}\ \textit{R-Field}$$

$$\frac{C <: D}{(D)\ \mathsf{new}\, C(\overline{v}) \mapsto \mathsf{new}\, C(\overline{v})}\ \textit{R-Cast}$$

$$\frac{\mathit{mbody}(m, C) = (\overline{x}, e_0)}{\mathsf{new}\, C(\overline{v_f}).m(\overline{v}) = [\overline{v}/\overline{x}, \mathsf{new}\, C(\overline{v_f})/\mathtt{this}]e_0}\ \textit{R-Invk}$$

$$\frac{\mathit{mbody}(m, C) = (\overline{x}, e_0)}{v.C::m(\overline{v}) = [\overline{v}/\overline{x}, v/\mathtt{this}]e_0}\ \textit{R-BoundInvk}$$

$$\frac{e \mapsto e'}{\mathit{Ctx}[e] \mapsto \mathit{Ctx}[e']}\ \textit{R-Context}$$

Figure 6: Dynamic Semantics

a non-terminating method. This causes the fragile base class problem, because the `addAll` method of `EffectfulCons` will behave differently depending on which superclass implementation is used. In the first implementation, `addAll` invokes `add` and will therefore continue executing indefinitely. In the second implementation, `addAll` does not use `add` and so it will terminate normally.

In Selective Open Recursion, these two implementations are in fact indistinguishable, because `add` is not `open` and so the call to `add` within `addAll` will always execute the `List`'s implementation of `add`.

Featherweight OpenJava also doesn't model other important modularity constructs such as `private` or `protected` methods or fields. While these constructs are important for information hiding, we omit them because they are orthogonal to the fragile base class problem.

The main technical result of this paper is a formal definition of *local equivalence* of two implementations, along with a proof that locally equivalent implementations cannot be distinguished by any client code. This result guarantees that as long as `open` methods are used consistently in different implementations, the fragile base class problem cannot occur.

5.3 Reduction Rules

The evaluation relation, defined by the reduction rules given in Figure 6, is of the form $e \mapsto e'$, read "expression e reduces to expression e' in one step." We write $\mapsto^*$ for the reflexive, transitive closure of $\mapsto$.

Most of the reduction rules are taken directly from Featherweight Java. The *R-Field* rule looks up the value of field f_i by returning the ith argument to the object constructor. As in Java (and FJ), the *R-Cast* rule checks that the cast expression is a subtype of the cast type.

The method invocation rule *R-Invk* uses the *mbody* helper function (defined in Figure 10) to determine the correct method body to invoke. The method invocation is replaced with the appropriate method body. In the body, all occur-

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D \ldots}{C <: D}\ \textit{Subtype-Class}$$

$$\frac{}{C <: C}\ \textit{Subtype-Reflex}$$

$$\frac{C <: D \qquad D <: E}{C <: E}\ \textit{Subtype-Trans}$$

Figure 7: Subtyping Rules

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textit{T-Var}$$

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

$$\frac{\Gamma \vdash e_0 : C_0 \qquad fields(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i : C_i}\ \textit{T-Field}$$

$$\frac{\Gamma \vdash e : D}{\Gamma \vdash (C)\ e : C}\ \textit{T-Cast}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : C_0 \qquad \Gamma \vdash \overline{e} : \overline{C} \\ mtype(m, C_0) = \overline{D} \to C \qquad \overline{C} <: \overline{D}\end{array}}{\Gamma \vdash e_0.m(\overline{e}) : C}\ \textit{T-Invk}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : C_0 \qquad \Gamma \vdash \overline{e} : \overline{C} \\ mtype(m, E) = \overline{D} \to C \qquad \overline{C} <: \overline{D} \\ C_0 <: E\end{array}}{\Gamma \vdash e_0.E::m(\overline{e}) : C}\ \textit{T-BoundInvk}$$

Figure 8: Typechecking

rences of the formal method parameters and `this` are replaced with the actual arguments and the receiver, respectively. Here, the capture-avoiding substitution of values $\overline{v}$ for variables $\overline{x}$ in e is written $[\overline{v}/\overline{x}]e$. Finally, the rule *R-Context* allows reduction to proceed within an expression. The contexts, also shown in Figure 6, define the usual left-to-right order of evaluation.

New Constructs. The *R-BoundInvk* rule gives the semantics of static dispatch. Instead of looking up the method body in the receiver class, the rule looks up the body in the explicitly named class.

5.4 Typing Rules

FOJ's subtyping rules are given in Figure 7. Subtyping is derived from the immediate subclass relation given by the `extends` clauses in the class table CT. The subtyping relation is reflexive and transitive, and it is required that there be no cycles in the relation (other than self-cycles due to reflexivity).

Typing judgments, shown in Figure 8, are of the form $\Gamma \vdash e : C$, read, "In the type environment Γ, expression e has type C."

$$\frac{\begin{array}{c}\overline{M}\ OK\ in\ C \qquad fields(D) = \overline{D}\ \overline{g} \\ K = C(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\ \{\ super(\overline{g}); \texttt{this}.\overline{f} = \overline{f};\ \}\end{array}}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \}\ OK}\ \textit{ClassOK}$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\ \texttt{extends}\ D \ldots \\ override(m, D, \overline{C} \to C_0) \\ \{\overline{x} : \overline{C},\ \texttt{this} : C\} \vdash e_0 : E_0 \qquad E_0 <: C_0\end{array}}{C_0\ m(\overline{C}\ \overline{x})\ \{\ \texttt{return}\ e_0;\ \}\ OK\ in\ C}\ \textit{MethOK}$$

Figure 9: Class and Method Typing

The *T-Var* rule looks up the type of a variable in Γ. The object creation rule verifies that the parameters to the constructor have types that match the types of that class's fields. The rule for field reads looks up the declared type of the field using the $fields$ function defined in Figure 10. The cast rule simply checks that the expression being cast is well-typed; a run-time check will determine if the value that comes out of the expression matches the type of the cast. For simplicity, we depart from Java and Featherweight Java by not explicitly modeling *stupid cast* errors [9].

Rule *T-Invk* looks up the invoked method's type using the $mtype$ function defined in Figure 10, and verifies that the actual argument types are subtypes of the method's argument types.

New Constructs. The *T-BoundInvk* rule checks the type of statically-dispatched method calls. It is identical to the *T-Invk* rule for dynamic dispatch, except that it checks that the type of the receiver is a subtype of the class C to which the method is dispatched.

5.5 Auxiliary Definitions

Figure 9 shows the rules for typing classes and methods. The typing rules for classes and methods have the form "class C is OK," and "method m is OK in C." The class rule checks that the methods in the class are well-formed, and that the constructor has the required form. The rule for methods checks that the method body is well typed, and uses the $override$ function (defined in Figure 10) to verify that methods are overridden with a method of the same type.

Figure 10 shows the definitions of many auxiliary functions used earlier in the semantics. These definitions are straightforward and in most cases are derived directly from rules in Featherweight Java. The $fields$ function looks up the field declarations in the class and adds them to the declarations inherited from the superclass.

The $mtype$ function looks up the type of a method in the class; if the method is not present, it looks in the superclass instead. Finally, the $override$ function verifies that if a superclass defines method m, it has the same type as the definition of m in a subclass.

New Constructs. The $open$ predicate is true only for methods that are declared with the `open` keyword. The $mbody$ function looks up the body of a method in much the same way as $mtype$ does, but it uses the $body$ function defined in

$$\frac{}{fields(\texttt{Object}) = \bullet}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \qquad fields(D) = \overline{D}\ \overline{g}}{fields(C) = \overline{D}\ \overline{g}, \overline{C}\ \overline{f}}$$

$$\frac{CT(C) = \texttt{class}\ C\ \ldots\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \qquad (D\ m(\overline{D}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mtype(m, C) = \overline{D} \to D}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \qquad m\ is\ not\ defined\ in\ \overline{M}}{mtype(m, C) = mtype(m, D)}$$

$$\frac{(mtype(m, C) = \overline{E} \to E) \implies (\overline{D} = \overline{E} \wedge D = E)}{override(m, C, \overline{D} \to D)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \qquad open\ E\ m(\overline{E}\ \overline{x})\{\ return\ e;\ \} \in \overline{M}}{open(C, m)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \ldots\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M};\ \} \qquad (D\ m(\overline{D}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mbody(m, C) = (\overline{x},\ body(C, e))}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \qquad m\ is\ not\ defined\ in\ \overline{M}}{mbody(m, C) = mbody(m, D)}$$

Figure 10: Auxiliary Definitions

Figure 11 in order to implement the static-dispatch semantics of Selective Open Recursion. The *body* function leaves most expressions unchanged, including object creation, field accesses, and casts. Method calls are left unchanged if the receiver is not `this`, or if the method being called is `open`. If the call is to a non-`open` method invoked on the current object `this`, the call is replaced with a call that is statically dispatched to the current class C.

5.6 Type Soundness

The type soundness theorems for our variant of Featherweight Java are stated exactly as in the original system. The only differences are additional cases in the progress and preservation theorems to handle statically bound method invocations. We omit the full proofs because they are tedious and similar to the original system.

Theorem 1 (Type Preservation)
If $\emptyset \vdash e : C$ and $e \mapsto e'$, then there exists a $D <: C$ such that $\emptyset \vdash e' : D$.

Proof: By induction over the derivation of $e \mapsto e'$. ■

$$\frac{\overline{e'} = body(C, \overline{e})}{body(C,\ \texttt{new}\ D(\overline{e})) = \texttt{new}\ D(\overline{e'})}\ \textit{Body-New}$$

$$\frac{body(C, e) = e'}{body(C,\ e.f) = e'.f}\ \textit{Body-Field}$$

$$\frac{body(C, e) = e'}{body(C,\ (D)\ e) = (D)\ e'}\ \textit{Body-Cast}$$

$$\frac{(e \neq \texttt{this} \vee open(C, m)) \qquad body(C, e) = e' \qquad \overline{e'} = body(C, \overline{e})}{body(C,\ e.m(\overline{e})) = e'.m(\overline{e'})}\ \textit{Body-Meth}$$

$$\frac{\overline{e'} = body(C, \overline{e}) \quad \neg open(C, m)}{body(C,\ \texttt{this}.m(\overline{e})) = \texttt{this}.C{::}m(\overline{e'})}\ \textit{Body-Self}$$

Figure 11: Method Body Translation

Theorem 2 (Progress)
If $\emptyset \vdash e : C$ then either

- *e is a value, or*
- *$e = Ctx[(D)\ \textit{new}\ E(\overline{v})]$ with $E \not<: D$, or*
- *$e \mapsto e'$ for some e'.*

Proof: By induction over the derivation of $\emptyset \vdash e : C$. ■

5.7 Local Equivalence

We would like to prove an abstraction property, stating that clients cannot distinguish between two different but semantically equivalent implementations of a library. To lay the groundwork for the abstraction property, we define a local *observational equivalence* relation that defines a precise notion of equivalence between two library implementations.

The rules for local observational equivalence are given in Figure 12. Two class tables are equivalent if they define the same set of class names and the implementations for corresponding class names are observationally equivalent. Two class implementations are equivalent if they have the same superclass name, the same sets of fields and method names, and their corresponding method implementations are observationally equivalent. Two methods are observationally equivalent if, for all possible arguments to the method, the method bodies substituted with the appropriate arguments behave in an observationally equivalent way.

Two expressions are observationally equivalent if they execute in bisimilar ways with respect to dynamically dispatched method calls. More specifically, two expressions are equivalent if they both reduce to the same value (rule *Eq-Val*) or if they both result in cast errors (rule *Eq-Error*), or if they both diverge (rule *Eq-Diverge*). While they are executing (towards a value, error, or divergence), they can execute in any way as long as they always dynamically dispatch to the same functions at corresponding points in their execution. We formalize this execution requirement with three rules.

Rule *Eq-Stat* allows the two expressions to each take any number of steps *excluding* the dynamic dispatch rule *R-Invk*,

$$\frac{domain(CT_1) = domain(CT_2) \quad \forall C \in domain(CT_1) \ : \ (CT_1, C) \simeq (CT_2, C)}{CT_1 \simeq CT_2} \ \textit{Eq-CT}$$

$$\frac{\begin{array}{c} CT_1(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \\ CT_2(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M'}\ \} \\ |M| = |M'| \\ \forall i \in \{1..|M|\}\ .\ (CT_1, D, M_i) \simeq (CT_2, D, M'_i) \end{array}}{(CT_1, C) \simeq (CT_2, C)} \ \textit{Eq-Cls}$$

$$\frac{\begin{array}{c} \forall \overline{v}\ \textit{such that}\ \emptyset \vdash \overline{v} : \overline{C}\ . \\ (CT_1, [\overline{v}/\overline{x}]body(D, e)) \\ \approx (CT_2, [\overline{v}/\overline{x}]body(D, e')) \end{array}}{\begin{array}{c} (CT_1, D, C\ m(\overline{C}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \\ \simeq (CT_2, D, C\ m(\overline{C}\ \overline{x})\ \{\ \texttt{return}\ e';\ \}) \end{array}} \ \textit{Eq-Meth}$$

$$\frac{}{(CT_1, v) \approx (CT_2, v)} \ \textit{Eq-Val}$$

$$\frac{E \not<: D \qquad E' \not<: D'}{\begin{array}{c} (CT_1, Ctx_1[(D)\ \texttt{new}\ E(\overline{v})]) \\ \approx (CT_2, Ctx_2[(D')\ \texttt{new}\ E'(\overline{v'})]) \end{array}} \ \textit{Eq-Error}$$

$$\frac{\begin{array}{c} CT_1 \vdash e_1 \overset{stat^*}{\mapsto} e'_1 \\ CT_2 \vdash e_2 \overset{stat^*}{\mapsto} e'_2 \\ (CT_1, e'_1) \approx (CT_2, e'_2) \end{array}}{(CT_1, e_1) \approx (CT_2, e_2)} \ \textit{Eq-Stat}$$

$$\frac{\begin{array}{c} (CT_1, Ctx_1 : C) \approx (CT_2, Ctx_2 : C) \\ CT_1; \emptyset \vdash v.m(\overline{v}) : C \end{array}}{(CT_1, Ctx_1[v.m(\overline{v}]) \approx (CT_2, Ctx_2[v.m(\overline{v}])} \ \textit{Eq-Dispatch}$$

$$\frac{\begin{array}{c} e_1\ \textit{and}\ e_2\ \textit{diverge according to} \\ \textit{Eq-Stat and Eq-Dispatch} \end{array}}{(CT_1, e_1) \approx (CT_2, e_2)} \ \textit{Eq-Diverge}$$

$$\frac{\begin{array}{c} \forall v'\ \textit{such that}\ \emptyset \vdash v' : C'\ \textit{and}\ C' <: C\ . \\ (CT_1, Ctx_1[v']) \approx (CT_2, Ctx_2[v']) \end{array}}{(CT_1, Ctx_1 : C) \approx (CT_2, Ctx_2 : C)} \ \textit{Eq-Context}$$

Figure 12: Local Equivalence

as long as the resulting two expressions are also observationally equivalent. The $\overset{stat^*}{\mapsto}$ relation thus represents the $\mapsto^*$ relation without the *R-Invk* rule.

Rule *Eq-Dispatch* allows two observationally equivalent expressions to take a single *R-Invk* reduction with method calls that are identical in the receiver values, the argument values, and the name of the methods called. The equivalence definition in Figure 12 is *local* in that it does not model the the execution of the called method in the observational equivalence framework. Since the called method might be defined in client code, not in the library itself, we cannot directly model its behavior using only the local definitions in the library. Instead, we ensure that for all values the method could possibly return (as determined by the return type C of the method), the contexts will execute with that value in observationally equivalent ways. Equivalence of contexts is formally defined in the last rule, *Eq-Context*.

$$\frac{}{(CT_1, v) \cong (CT_2, v)} \ \textit{Bi-Val}$$

$$\frac{E \not<: D \qquad E' \not<: D'}{\begin{array}{c} (CT_1, Ctx_1[(D)\ \texttt{new}\ E(\overline{v})]) \\ \cong (CT_2, Ctx_2[(D')\ \texttt{new}\ E'(\overline{v'})]) \end{array}} \ \textit{Bi-Error}$$

$$\frac{\begin{array}{c} CT_1 \vdash e_1 \overset{stat^*}{\mapsto} e'_1 \\ CT_2 \vdash e_2 \overset{stat^*}{\mapsto} e'_2 \\ (CT_1, e'_1) \cong (CT_2, e'_2) \end{array}}{(CT_1, e_1) \cong (CT_2, e_2)} \ \textit{Bi-Stat}$$

$$\frac{\begin{array}{c} CT_1 \vdash Ctx_1[v_1.m(\overline{v_1})] \mapsto e'_1 \\ CT_2 \vdash Ctx_2[v_2.m(\overline{v_2})] \mapsto e'_2 \\ (CT_1, e'_1) \cong (CT_2, e'_2) \end{array}}{(CT_1, Ctx_1[v_1.m(\overline{v_1})]) \cong (CT_2, Ctx_2[v_2.m(\overline{v_2})])} \ \textit{Bi-Dispatch}$$

$$\frac{\begin{array}{c} e_1\ \textit{and}\ e_2\ \textit{diverge according to} \\ \textit{Bi-Stat and Bi-Dispatch} \end{array}}{(CT_1, e_1) \cong (CT_2, e_2)} \ \textit{Bi-Diverge}$$

Figure 13: Global Equivalence

5.8 Global Equivalence

We wish to prove that given any two observationally equivalent libraries, any client code will behave equivalently when run against these two libraries. Figure 13 defines a notion of equivalence that is more global than local equivalence in that it tracks the execution of an expression through dynamic dispatch. Global equivalence is defined in the same way as local equivalence in rules *Bi-Val*, *Bi-Error*, *Bi-Stat*, and *Bi-Diverge*. However, the rule for dynamic dispatch checks that the actual bodies of the called methods execute in bisimilar ways, rather than just ensuring that the result of the method call will be handled in equivalent ways. Bisimulation thus provides a complete, global picture of whether two clients behave identically or not.

5.9 Abstraction

The abstraction theorem states that if two libraries CT_1 and CT_2 are locally equivalent, then no matter what well-typed client code e we write, that client will execute in a globally equivalent way against both libraries. More formally:

Theorem 3 (Client Abstraction)
If $CT_1 \simeq CT_2$, then $\forall e$ such that $CT_1, \emptyset \vdash e : C$ and $CT_2, \emptyset \vdash e : C$, we have $(CT_1, e) \cong (CT_2, e)$.

Proof:

We prove the theorem by showing that execution of (CT_1, e) and (CT_2, e) preserves an invariant, and that all executions that are compatible with that invariant also conform to the definition of global equivalence.

The invariant states that the two executing expressions can be expressed in the form $Ctx_0(..(Ctx_n(e))..)$ and $Ctx'_0(..(Ctx'_n(e'))..)$, for some $n, e, e', Ctx_{0..n}, Ctx'_{0..n}$. We

also have $e \approx e'$, and $\forall i \in \{1..n\}$ we have $(Ctx_i : C_i) \approx (Ctx'_i : C_i)$ where $\emptyset \vdash Ctx_{i+1}(..(Ctx_n(e))..) : C_i$.

This invariant is initially true, with the expressions $e = e'$, and $n = 0$.

For any expression in the form described by the invariant, one of the following will be true, by the definition of $\approx$:

- Case *Eq-Val*: $e = e' = v$ and v is a value. If $n = 0$, then execution halts with the value v. Otherwise, we can reduce n by one, let $e = Ctx_n[v]$, and let $e' = Ctx'_n[v]$; by assumption these contexts are locally equivalent, so the new expressions e and e' are as well. Thus, the overall expression obeys the invariant, so we may continue.
- Case *Eq-Error*: Both e and e' have a cast error at the current locus of execution. Execution therefore halts with a cast error.
- Case *Eq-Stat*: In this case, e and e' each take one or more execution steps, without including a dynamic dispatch, resulting in another state where $e \approx e'$. The resulting expression obeys the overall invariant.
- Case *Eq-Dispatch*: In this case, e and e' reach a dynamic dispatched function call with the same receiver and argument values. We increase n by 1, and add the context of the function call to the stack of contexts, noting that the function call contexts must be locally equivalent according to the rule. The new expression becomes the body of the looked-up function in each class table, with the receiver and argument values substituted appropriately. Since the class tables were locally equivalent, the substituted method bodies e and e' must also be locally equivalent. Thus the whole expression obeys the invariant, and we may continue with execution.
- Case *Eq-Diverge*: Both e and e' diverge without making any further dynamic dispatches. The overall expression thus diverges in the same way.

In all the cases shown above, the invariant is maintained by execution.

Now, by inspection we note that repeated application of the cases shown above results in an execution sequence that is compatible with the rules for global equivalence. The overall expressions execute in lockstep with respect to dispatches, taking any number of non-dispatching steps in between. Execution either diverges according to this pattern, or it terminates in an error or in two equal values. Thus, the original expression e executes in a globally equivalent way against either class table.

■

We would like to strengthen the abstraction theorem to allow us to not only define an expression to evaluate in the context of two libraries, but also to add new classes that might be subclasses of classes in the two libraries. The library abstraction theorem states that if two libraries are locally equivalent, then no matter what well-typed classes we add that build on the library, these classes will be locally equivalent no matter which library they are compiled against. More formally:

Theorem 4 (Library Abstraction)
If $CT_1 \simeq CT_2$, then $\forall CT'$ such that $CT_1 \cup CT'$ OK and $CT_1 \cup CT'$ OK, we have $CT_1 \cup CT' \simeq CT_1 \cup CT'$.

Proof outline. For each method in CT', we apply reasoning similar to that in the Client Abstraction theorem to show that the method body expression e executes in an locally equivalent way regardless of what library it is compiled against. Since the method bodies are textually equal, it is unsurprising that they behave equivalently.

By assumption, all the methods and classes in CT_1 were locally equivalent to the corresponding structures in CT_2, and since the local equivalence rules are local and do not depend on any definitions not in CT_1 or CT_2, this property continues to hold in the extended systems. Thus, the extended class tables are also locally equivalent.

5.10 Applying Abstraction

We can apply the definition of local equivalence and the abstraction theorem to show that the two definitions of `List` in Figure 5 are indistinguishable by clients in our system. It is easy to show that the `addAll` methods are locally equivalent. The only difference in the methods is that the first method makes a statically-dispatched call to `add`, which immediately reduces to the body of the `add` function, which is exactly the code used in the second method. By the abstraction theorem, no client can distinguish the two implementations, i.e. they are globally equivalent. Thus the fragile base class problem cannot occur, and the developer of `List` is free to switch between these two implementations.

In contrast, under the standard Java semantics (where every method is treated as being `open`), these two implementations are *not* locally equivalent. This is because the call to `add` is dynamically dispatched in Java, allowing subclasses (such as `EffectfulCons`) to distinguish its behavior from the other implementation which does not make this dynamically dispatched call.

As this discussion implies, our abstraction theorem is true for plain Java programs under the assumption that every method is conceptually marked as `open`. However, the theorem is much less meaningful for plain Java, because so many implementation changes are prohibited by the abstraction theorem when every method is `open`. It is only the presence of non-open methods that allows a library developer to evolve the libraries implementation in significant ways.

In summary, this example together with the abstraction property shows how our solution avoids the fragile base class problem, allowing developers to make more changes to base class code, while still supporting open recursion semantics where they are needed.

6. Related Work

A significant body of related research focuses on documenting the dependencies between methods in a *specialization interface*. Kiczales and Lamping proposed that a method should document which methods it depends on, so that subclasses can make accurate assumptions about the superclass implementation [10]. Steyaert et al. propose a similar approach in a more formal setting [15]. Ruby and Leavens suggest documenting method call dependencies as part of a broader focus on modular reasoning in the presence of inheritance [14]. They also document a number of design guidelines that are applicable to the setting of Selective Open Recursion.

A common weakness of the "dependency documentation" approaches described above is that they solve the fragile base class problem not by hiding implementation details,

but rather by exposing them. Since the calling patterns of a class are part of the subclassing interface–and since subclasses may depend on them–making significant changes to the implementation of the class become impossible. Steyaert et al. acknowledge this and suggest documenting only the "important method calls," but the fragile base class problem can still occur unless unimportant method calls are hidden from subclasses using a technique like ours. Our work requires that calling patterns be maintained for calls to `open` methods, but does not impose this requirement for non-open methods, allowing a much wider range of implementation changes.

Bloch, Szyperski, and others suggest using delegation in place of inheritance as a way of avoiding the fragile base class problem [4, 16]. However, as Szyperski notes, not all uses of inheritance can be replaced by delegation because open recursion is sometimes needed [16]. Selective Open Recursion provides a middle ground between inheritance and delegation, providing open recursion when it is needed but the more modular delegation semantics where it is not.

Mikhajlov and Sekerinski consider a number of different ways in which an incorrect use of inheritance can break a refinement relationship between a class and its subclasses [11]. They prove a flexibility theorem showing that under certain conditions, when a superclass C is replaced with a new implementation D, then C's subclasses still implement refinements of the original implementation C. Their results, however, do not appear to guarantee that the semantics of C's subclasses are unaffected by the new implementation D, which is the contribution of our work.

A proof of an abstraction property for a full object-oriented language must not only deal with the fragile base class problem, but also with the problems caused by shared mutable state. Banerjee and Naumann have shown how abstraction can be enforced in the presence of mutable state by encapsulating the state within an object using ownership [3]. Their notion of ownership is quite restrictive; we are currently trying to generalize their result to more flexible ownership systems such as Ownership Domains [2]. Recent work on separation logic also shows promise for modular reasoning in the presence of shared mutable state [12].

Our use of static dispatch for calls on `this` is related to the *freeze* operator provided by module systems such as Jigsaw [6]. The freeze operation statically binds internal uses of a module declaration, while allowing module extensions to override external uses of that declaration.

Our solution to the fragile base class problem was inspired by our earlier work on a related modularity problem in aspect-oriented programming [1]. Just as a `CountingSet` subclass of `Set` can observe whether `addAll` is implemented in terms of `add`, a `Counting` aspect can be defined that uses advice to make the same observation. Our solution there was to prohibit aspects from advising internal calls within a class or module–just as we solve the fragile base class problem by using static dispatch to prevent subclasses from intercepting implementation-dependent calls in their superclass. In the aspect-oriented setting, we allow modules to export pointcuts that act as disciplined extension points, similar to `open` methods. Both abstraction proofs rely on a bisimilar execution with respect to pointcuts and `open` methods, respectively.

Relative to previous work, we believe ours is the first to address the fragile base class problem by distinguishing methods for which open recursion is needed from methods for which it is not. Ours is also the first to formally prove an abstraction property in the presence of open recursion, stating that subclasses are unaffected by semantics-preserving changes to the superclass.

7. Conclusion

This paper argued that the fragile base class problem occurs because current object-oriented languages do not distinguish internal method calls that are invoked for mere convenience from those that are invoked as explicit extension points for subclasses. We proposed to make this distinction explicit by labeling as `open` those methods to which open recursion should apply. We used a formal model to prove, for the first time, an abstraction property for objects stating that locally equivalent implementations of a library cannot be distinguished by that library's clients. Our results mean that library designers can freely change many aspects of a library's implementation without the danger of breaking subclass code.

8. Acknowledgments

I thank Craig Chambers, Todd Millstein and Frank Pfenning for their feedback on earlier drafts of this material.

9. References

[1] J. Aldrich. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. In *Foundations of Aspect Languages*, March 2004.

[2] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming*, June 2004.

[3] A. Banerjee and D. A. Naumann. Representation Independence, Confinement, and Access Control. In *Principles of Programming Languages*, January 2002.

[4] J. Bloch. *Effective Java*. Addison-Wesley, Reading, Massachusetts, 2001.

[5] B. Bokowski and A. Spiegel. Barat–A Front-End for Java. Freie Universitt Berlin Technical Report B-98-09, 1998.

[6] G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. Ph.D. Thesis, Dept. of Computer Science, University of Utah, 1992.

[7] E. Ernst. Family Polymorphism. In *European Conference on Object-Oriented Programming*, June 2001.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] A. Igarashi, B. Pierce, and P. Wadler. Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[10] G. Kiczales and J. Lamping. Issues in the Design and Documentation of Class Libraries. In *Object-Oriented Programming Systems, Languages, and Applications*, 1992.

[11] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *European Conference on Object-Oriented Programming*, 1998.

[12] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and Information Hiding. In *Principles of Programming*

Languages, January 2004.

[13] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[14] C. Ruby and G. T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[15] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1996.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.