Statically Scoped Object Adaptation with Expanders

Alessandro Warth Milan Stanojević Todd Millstein
Computer Science Department
University of California, Los Angeles
{awarth,milanst,todd}@cs.ucla.edu

ABSTRACT

This paper introduces the *expander*, a new object-oriented (OO) programming language construct designed to support object adaptation. Expanders allow existing classes to be noninvasively updated with new methods, fields, and superinterfaces. Each client can customize its view of a class by explicitly importing any number of associated expanders. This view then applies to all instances of that class, including objects passed to the client from other components. A form of *expander overriding* allows expanders to interact naturally with OO-style inheritance.

We describe the design, implementation, and evaluation of eJava, an extension to Java supporting expanders. We illustrate eJava's syntax and semantics through several examples. The statically scoped nature of expander usage allows for a modular static type system to prevent several important classes of errors. We describe this modular static type system informally, formalize eJava and its type system in an extension to Featherweight Java, and prove a type soundness theorem for the formalization. We also describe a modular compilation strategy for expanders, which we have implemented using the Polyglot extensible compiler framework. Finally, we illustrate the practical benefits of eJava by using this implementation in two case studies.

1. INTRODUCTION

Inheritance in object-oriented (OO) languages provides a form of extensibility for classes. A client of a class `C` can use inheritance to easily create a customized version of `C` without requiring source-code access to or recompilation of `C`. Further, the modular static type systems in mainstream languages like Java and C# ensure that this subclass can be safely used where `C` was expected.

However, several programming scenarios require forms of *object adaptation*, and today's OO languages do not easily support this idiom. For example, consider a database of business objects, which a particular client wishes to display in an application-specific manner. The client likely needs to add new methods to these objects, and possibly to make these objects meet a client-specific interface. Inheritance would allow a new class to have these new behaviors but would do nothing to augment the existing objects. Design patterns like *adapter* and *visitor* [14] provide protocols that can be used to achieve the desired extensibility. However, these patterns are often tedious and error prone to implement, can require advance planning on the part of the original implementer of the class being extended, require invasive modifications whenever the adapted class hierarchy is augmented, and often rely on statically unsafe constructs like type casts.

In this paper, we introduce the *expander*, a new language construct that provides explicit support for object adaptation. An expander is a repository for adding new behaviors to existing classes and can include new methods, fields, and superinterfaces. Clients of an expander may make use of these additional behaviors on any object of the original class (or a subclass), including objects passed to the client from other components. Expanders also interact in a natural way with inheritance: expanders can be overridden for particular subclasses, and message sends dynamically dispatch to the best method for the given receiver, as with traditional message sends.

There has been a large body of research into mechanisms for increasing the extensibility of classes. Expanders are distinguished by a novel combination of design properties that are targeted to support object adaptation:

In-place adaptation. An expander updates an existing class with new behaviors, rather than creating a new class that contains the additional behaviors. This design choice is critical for support of object adaptation. In particular, the new behaviors in an expander are available for use on instances of the original class, regardless of where or when those instances were created.

This design contrasts with the variety of proposals for flexible class extensibility, including mixins [5, 12], traits [24], proposals that include a form of dependently typed classes [9, 10, 19, 21], and parameterized module systems for OO languages [11, 18]. These works have a very different goal from ours and are quite complementary. The customizations supported by these languages provide for fine-grained forms of code reuse across classes and class hierarchies but do nothing to adapt existing objects. Indeed, typically some advance planning, for example via the factory design pattern [14] or a form of module parameterization, is required to ensure that the full customization of a class can be produced before any instances are created. In contrast, expanders support object adaptation and allow multiple clients to view the same object in different ways, but they do not provide code reuse across classes and class hierarchies.

Statically scoped adaptation. Expanders target the problem of adapting existing classes to the needs of new clients, without affecting the behavior of existing clients. This focus led us to a design whereby each client explicitly imports the expanders that are needed in order to perform its task; all other expanders are out of

scope and cannot affect the client's behavior. In addition to making each client's behavior easier to reason about, this static scoping of expanders naturally allows an object to be "expanded" in different ways by different clients in the same program without any conflict.

This design contrasts with the inter-type declarations in aspect-oriented languages like AspectJ [17], which also allow existing classes to be augmented with new methods, fields, and superinterfaces. While aspects can be used for object adaptation, such adaptation affects all clients of a class implicitly, including existing ones. While such power can be quite useful, it also makes it difficult to understand how a given client will behave without global knowledge of all aspects in the program.

Modular type safety. The existence of multiple versions of a class has the potential to cause erroneous or undesirable behavior at run time, whereby a field or method of one version is expected but another is actually used. This problem is exacerbated in the presence of inheritance, since subclasses of the original class may add behaviors that clash with those added by an expander. Aside from violating programmer expectations, these kinds of problems also potentially compromise type safety, for example if a field of one type is expected but a field of a different type is actually used.

Because of the static scoping of expanders as described above, it is possible to perform modular static checking to guarantee type safety in the presence of expanders. The scoped nature of expanders ensures that the run time behavior of a class that passes the static checks will be the same regardless of what unknown classes and expanders are ultimately linked into the final application. To our knowledge, this guarantee of modular type safety is unique among languages that are expressive enough to support object adaptation, including the AspectJ language described above and the recent work on Classboxes [4, 3].

We have instantiated our notion of expanders in the language *eJava*, a backward-compatible extension to Java. We have designed a modular type system for eJava as well as a modular compilation strategy from eJava to Java, and we have implemented both in the Polyglot extensible compiler framework [20]. We have also formalized eJava as an extension of Featherweight Java (FJ) [16] that we call Featherweight eJava (FeJ), and we have proven its modular type system sound.

Finally, we have used our eJava implementation to gain experience with the language and gauge its practical utility. First, we have used eJava to build a small application that employs Java's Swing library to display some existing business objects in an application-specific manner, and we compare the benefits and limitations against an implementation in pure Java. Second, we have performed an exploratory study on the Eclipse integrated development environment [7]. We identify several Java-based extensibility idioms used in the implementation of Eclipse, and we explore the ways in which expanders can allow these idioms to be more naturally and reliably expressed.

The rest of the paper is structured as follows. Section 2 introduces expanders by a number of examples. Section 3 details eJava's method-lookup semantics and associated static typechecks. Section 4 describes the FeJ formalism, which provides a more precise description of the language and its modular type system. Section 5 describes the modular compilation strategy for eJava, as implemented in the eJava compiler. Section 6 discusses our two experiments with the language. Section 7 compares expanders with related work, and Section 8 concludes.

2. EXPANDERS

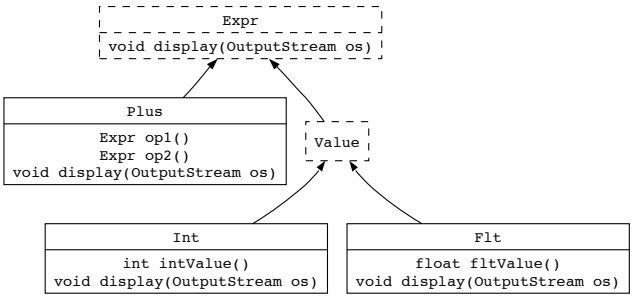

Figure 1: An expression hierarchy.

This section informally describes the capabilities of expanders and their interaction with the other features of Java. As a running example, consider a `Parser` class that parses a simple language of expressions supporting integers, floating-point numbers, and addition. The `Parser` has a `parse` method that accepts a `String` and produces an abstract syntax tree (AST), which is represented using the `Expr` class hierarchy shown in Figure 1. We illustrate eJava's expressiveness and flexibility by considering a number of ways in which clients may wish to employ this parser as part of a larger application. To reuse the parser, clients must first customize it to their needs. eJava allows each client's customizations to be expressed declaratively, without modifying or requiring access to the source code of the parser or the `Expr` hierarchy, without requiring any advance planning by the implementer of the parser, and without interfering with the customizations of other clients.

2.1 Noninvasive Visitors with Expanders

It is natural for clients to wish to augment the `Expr` hierarchy with new methods, in order to perform new passes over the AST. For example, suppose a client wishes to add an `eval` method that evaluates an expression and returns the resulting value. One approach would be to create a subclass `EvalExpr` of `Expr` that includes such a method, as well as new classes like `EvalPlus` and `EvalInt`. However, the lack of multiple implementation inheritance in Java would require this approach to duplicate a significant amount of code. For example, `EvalPlus` must inherit from `EvalExpr` in order to override its `eval` method, but then it cannot also inherit from `Plus`, forcing the client to duplicate all of the existing capabilities of addition expressions. Worse, this approach would not allow the existing `Parser` class to be reused, since it creates instances of the original expression classes rather than the new ones.

The standard approach to this problem is the *visitor* design pattern [14], which allows clients to easily add new functionality to existing classes externally. However, this approach requires the original implementer of the expression hierarchy to have anticipated the need for visitors, by providing a `Visitor` class and the appropriate `accept` methods for each kind of expression. The visitor pattern also has other, more minor, problems, for example the fact that every external operation must have the same argument and result types.

Therefore, clients are often forced to add new functionality using *utility classes* such as `Evaluator`, shown in Figure 2. While this approach allows the "method" to be added without modifying existing code, it has several drawbacks. Since the method is placed in the `Evaluator` class, dynamic dispatch on the `Expr` instance must be performed manually via runtime type tests and type casts. These type tests must be executed in the correct order (from most- to least-specific), which is easy to get wrong for large and com-

```
// file Evaluator.java
package eval;
import ast.*;
public class Evaluator {
  public static Value eval(Expr e) {
    if (e instanceof Value)
      return (Value)e;
    else if (e instanceof Plus) {
      Plus p = (Plus)e;
      Value v1 = eval(p.op1());
      Value v2 = eval(p.op2());
      // ...
    }
    else throw new EvalError();
  }
}
```

Figure 2: An external evaluator for `Expr`s in Java.

```
// file Calculator.java
package calc;
import parse.Parser;
import ast.*;
import eval.Evaluator;
public class Calculator {
  public void process(String s) {
    Expr e = new Parser().parse(s);
    Value ans = Evaluator.eval(e);
    ans.display(System.out);
  }
}
```

Figure 3: A client of the evaluator in Figure 2.

plex type hierarchies. Further, using utility classes like `Evaluator` forces clients to distinguish syntactically between calls to external methods of `Expr` like `eval` and calls to ordinary methods of the hierarchy like `display`. This difference is illustrated by the sample client in Figure 3, which implements a calculator application using `Parser` and `Evaluator`.

Expanders in eJava provide a natural solution to these problems. An expander is a repository for augmenting an existing class in several different ways, including the introduction of new methods, without requiring source-code access to the original class. For example, an expander that adds an `eval` method to `Expr` would look as follows:

```
expander EX of Expr {
  public Value eval() {...}
}
```

This expander is given the name `EX` and is declared to expand the `Expr` class. The body of the expander then provides the new `eval` method, just as it would be declared in the original class. For example, the method has an implicit receiver argument of type `Expr` which can be referenced as `this` in the method's body. An expander can declare any number of new methods.

We could choose to implement the semantics of expression evaluation entirely within the `eval` method of the above expander. This implementation would perform manual type tests and casts on `this` to determine how evaluation should proceed, analogous to the code in Figure 2. However, expanders support a better solution through

```
// file EX.ej
package eval;
import ast.*;
public expander EX of Expr {
  public Value eval() { throw new EvalError(); }
}
public expander EX of Value {
  public Value eval() { return this; }
}
public expander EX of Plus {
  public Value eval() {
    Value v1 = op1().eval();
    Value v2 = op2().eval();
    // ...
  }
}
```

Figure 4: An external evaluator for `Expr`s in eJava.

expander overriding, as shown in Figure 4. In addition to the expander for `Expr`, the compilation unit (i.e., file) shown includes two expanders that provide overriding implementations of `eval` for particular subclasses of `Expr`. We say that the three expanders are in the same *expander family*, which is a collection of expanders of the same name. Each expander family has a unique *top expander*: all other expanders in the family augment a type that is a subclass of the top expander's type. Therefore, the first expander in the figure is the top expander for `EX`.

The use of expander overriding obviates the need for `instanceof` tests and type casts. Instead, the method implementations make use of Java's ordinary dynamic dispatch on the receiver argument, just as if they were declared in the original expression classes.

We allow methods in expanders to override methods in other expanders, as the example shows. However, a method in an expander cannot override an ordinary method of the associated class. This restriction is the product of a tradeoff between modular compilation and external method overriding that has been previously recognized [6].[1] Further, we disallow overriding expanders from declaring members that are not declared in the top expander. This simplification ensures that an expander family presents a unique "interface" to clients, and results in no loss of expressiveness since clients can always create another expander for a particular subclass of the top expander's associated class.

An expander does not have privileged access to its associated receiver class, instead obeying the same visibility rules as other clients of the class. This restriction ensures that an expander does not break any internal invariants of existing classes. An expander as well as its members can have the usual Java privacy modifiers. These modifiers are interpreted relative to the expander's location, rather than that of the original class. For example, a method with package visibility in an expander is only accessible within the package in which that expander (not the class it expands) was declared.

Figure 5 shows the new version of our calculator application that uses `EX` (instead of `Evaluator`). In order to make use of the "expanded" view of the expression hierarchy, the expander family `EX` must be explicitly `used`, with a syntax analogous to Java's `import` statement. The methods declared for the `EX` expander family are thereby made available for use, via the same syntax as if they were

[1] Expanders are still allowed to *overload* methods of their associated classes.

```
// file Calculator.ej
package calc;
import parse.Parser;
import ast.*;
use eval.EX;
public class Calculator {
  public void process(String s) {
    Expr e = new Parser().parse(s);
    Value ans = e.eval();
    ans.display(System.out);
  }
}
```

Figure 5: A client of the evaluator in Figure 4.

declared in the original classes. This is illustrated by the invocation `e.eval()` in the figure. Without the `use` statement, that invocation would fail to typecheck, since `Expr` does not originally support an `eval` method.

The explicit import of expanders via `use` is critical for supporting safe modular reasoning about clients. For example, any client of the expression hierarchy that does not `use EX` cannot be affected by the methods introduced by the `EX` expander family. Similarly, there can exist multiple expander families that augment the expression hierarchy with an `eval` method. Different clients in the same program can `use` the version of expression evaluation that suits their needs, without interfering with other clients. The rules for method lookup, as well as our static typechecks to ensure that method lookup will always succeed at run time, are described in Section 3.

An expander can be used to add functionality to any class, including `final` classes. For example, one could implement an expander of Java's `String` class that adds a helper method like `removeWhitespace()`. Clients can then `use` this expander and invoke `removeWhitespace` on any value of type `String`, including string literals. Because an expander does not have privileged access to the class being expanded, this expander cannot violate any of the internal invariants of Java's `String` class.

2.2 Adapting with Expanders

While the ability to add methods to existing classes from the outside is very useful, often it is not enough on its own. For example, suppose that our `Calculator` class is to be used within an application in which all console output must be enqueued in a printing thread via the `enqueue()` method of the `Printer` class:

```
public static void enqueue(Printable p) {...}
```

This method takes an instance of the `Printable` interface, which is declared as follows:

```
public interface Printable { void print(); }
```

In Java, a programmer could use the *adapter* design pattern [14] to adapt existing instances of `Expr` and subclasses to support the `Printable` interface. To do so, the programmer would create a wrapper class `PrintableExpr` for `Expr` that meets the `Printable` interface and implements the associated methods by appropriately forwarding to the methods of the underlying expression. Whenever an instance of `Expr` or a subclass needs to be treated as `Printable`, it would first be adapted by explicitly constructing an instance of `PrintableExpr`, passing the adaptee object as a parameter.

Expanders provide a simpler and more natural solution to this problem, as shown in Figure 6. An expander can use the

```
// file PX.ej
package exprAdapt;
import ast.*;
import printer.Printable;
public expander PX of Expr implements Printable {
  public void print() { display(System.out); }
}
```

Figure 6: Adapting `Expr`s to `Printable`s with expanders.

```
// file Calculator.ej
package calc;
import parse.Parser;
import ast.*;
import printer.Printer;
use eval.EX;
use exprAdapt.PX;
public class Calculator {
  public void process(String s) {
    Expr e = new Parser().parse(s);
    Value ans = e.eval();
    Printer.enqueue(ans);
  }
}
```

Figure 7: A fancier calculator, which uses both evaluation and printing functionality.

`implements` clause to declare that the class being expanded (and its subclasses) meets any number of new interfaces. The interface can be implemented by a combination of methods already declared or inherited in the original class and methods declared in the expander. Overriding expanders can also be used in conjunction with the ability to meet new interfaces. For example, we could provide an overriding expander to define `print` specially for `Plus` expressions.

Figure 7 shows an upgraded version of our `Calculator` class, which uses both evaluation and printing functionality. The compilation unit now `use`s both `EX` and `PX`, allowing `Expr` instances to be viewed as containing both `eval` and `print` methods, as well as meeting the `Printable` interface. Therefore, the `e` instance of `Expr` is a typesafe argument to `enqueue`, without requiring any explicit adaptation as would be necessary with the adapter design pattern. It is important to note that the implementer of the `Printer` class need not know about the existence of either the `Expr` hierarchy or its `PX` expander. All that the `enqueue` method of `Printer` requires is that the given argument meets the `Printable` interface; our modular compilation strategy described in Section 5 ensures that this requirement is satisfied.

2.3 Adding State with Expanders

Expanders can also be used to add state to existing classes. An example is shown in Figure 8, which augments the expression hierarchy with type information. A `type` field is added to `Expr` (and subclasses) to store an expression's type, along with a getter method and a method to compute the type.

2.4 Expanding Interfaces

Expanders can also be used to add functionality to interfaces. For example, Figure 9 uses an expander to add a `printAll` method

```
// file TX.ej
package types;
import ast.*;
public expander TX of Expr {
  private Type type = null;
    public void typecheck() { type = new ErrType(); }
  public Type type() {
    if (type == null)
      typecheck();
    return type;
  }
}
public expander TX of Int {
  public void typecheck() { type = new IntType(); }
}
public expander TX of Flt {
  public void typecheck() { type = new FltType(); }
}
public expander TX of Plus {
  public void typecheck() {
    Type t1 = op1().type();
    Type t2 = op2().type();
    // ...
  }
}
```

Figure 8: Adding state in an expander.

```
// file IX.ej
import java.util.*;
public expander IX of Iterable {
  public void printAll() {
    Iterator it = iterator();
    while (it.hasNext())
      System.out.println(it.next());
  }
}
```

Figure 9: An expander for an interface.

to instances of Java's `Iterable` interface that prints all elements of the iterable collection. The expander makes use of the `iterator` method defined for `Iterable`.

An expander for an interface is not tied to a particular class hierarchy. Instead, it can be applied to an instance of any class that implements its associated interface. This interface acts as the abstract "requirements" of that expander, so the expander can be considered to be parameterized by the classes that it expands.

Further, other expanders can be used to increase the applicability of an interface-based expander. As shown earlier, an expander can adapt an existing class to support a new interface. This capability can therefore be used to allow an existing class to meet the requirements of an interface-based expander.

As an example, consider the task of implementing an alternative to the queue-based `Printer` class with an expander. Figure 10 shows `SPX`, the safe printing expander, which augments instances of the `Printable` interface with "safe-printing" functionality (the `safePrint` method uses the `Printable` class as a lock to ensure that no two `Printable` objects are ever printed to the console at the same time). Although `Expr` does not originally implement `Printable`, its instances can be augmented by `SPX` provided we first adapt them to the `Printable` interface by using `PX`, the `Printable` expander from Figure 6.

```
// file SPX.ej
package printer;
public expander SPX of Printable {
  public void safePrint() {
    synchronized (Printable.class) { print(); }
  }
}
```

Figure 10: `SPX`, the safe printing expander.

```
// file FancyCalculator.ej
package client;
import ast.*;
import fancyParse.Parser;
import fancyAST.Minus;
use eval.EX;
expander EX of Minus {
  public Value eval() {...}
}
public class FancyCalculator {
  public void process(String s) {
    Expr e = new Parser().parse(s);
    Value ans = e.eval();
    ans.display(System.out);
  }
}
```

Figure 11: Adding a new expander to an existing expander family.

2.5 Locally Augmenting Expander Families

In the examples so far, all expanders in a given family were declared in the same compilation unit. However, it is possible for an expander family's associated class hierarchy to be augmented by clients with new subclasses. When this happens, it is often desirable to augment the expander family with overriding expanders to provide special-purpose behavior for these classes. To this end, eJava allows a client of an expander family to *locally* augment that family with overriding expanders for new types. The additional expanders are *hygienic*: they only affect the expander family's behavior within the current compilation unit.

For example, suppose a client of the original expression hierarchy adds a subclass `Minus` of `Expr` and creates a new parser to handle subtraction. Figure 11 shows how a client of the `EX` expander can update it to handle `Minus` appropriately. The call to `eval` in `FancyCalculator` will dynamically dispatch to the new `eval` implementation whenever the receiver is an instance of `Minus`.

3. METHOD LOOKUP AND MODULAR TYPECHECKING

eJava's rules for method lookup are a natural generalization of those of Java. At compile time, each message send expression in a program is determined to refer to a unique *method family*, or else a static error is signaled. At run time, the most-specific method for the given receiver argument in the statically determined method family is selected and invoked. We discuss each of these phases

in turn, along with the corresponding static checks to ensure that method lookup always succeeds at run time. The section ends by illustrating a type hole that can arise in other languages for object adaptation, which our semantics for method lookup safely avoids.

3.1 Static Method Family Selection

Each method in eJava, and in Java, can be thought of as belonging to a unique method family. The method family of a method m_1 is determined as follows. If m_1 overrides a method m_2, then m_1 is in the same method family as m_2. Otherwise, m_1 belongs to a different method family. The Java typechecker statically ensures that there is a single best method family for each message send expression in a program, as discussed in the Java Language Specification [15], §15.12. In the presence of static overloading, a method family is essentially defined by the method's name, number of arguments, and static argument types. An error is signaled if either there are no applicable method families for a given message send or there are multiple applicable families but no most-specific one.

In eJava, we first use Java's rules to find a most-specific method family for a message send, completely ignoring expanders. If a unique such method family exists, we are done. If there are multiple such families but no most-specific one, then an ambiguity error occurs, as in Java. However, if there are no applicable method families, we then search for a unique applicable method family in the expanders that are `use`d by the current compilation unit, signaling a compile-time error otherwise. As in Java, the method family associated with a method in an expander is defined by the method's name, number of arguments, and static argument types, and additionally by the name of the enclosing expander. The rules for statically determining to which field declaration a field access refers are analogous, as are the rules that determine whether an object can be viewed as implementing a particular interface.

To illustrate these rules, consider statically determining the method family for the invocation `e.eval()` in Figure 5:

- Given the definitions of the `Expr` class (Figure 1) and the `EX` expander family (Figure 4), the invocation is statically determined to invoke the method family consisting of the three `eval` methods in the `EX` expanders.
- If the `use EX;` statement were omitted from Figure 5, the eJava typechecker would signal a static error, since there is no applicable method family for the invocation.
- If the original `Expr` class contained a (possibly `abstract`) zero-argument `eval` method, then that method's associated family would be selected, even in the presence of the `use EX;` statement.
- Suppose there existed another expander `EX2` that also defines a zero-argument `eval` method for `Expr`. If Figure 5 were augmented to include the statement `use EX2;`, the eJava typechecker would signal a static error, since the `eval` method families in `EX` and `EX2` are ambiguous. However, the error is only signaled because of the call to `eval`. If that call were removed, `EX` and `EX2` could both be `use`d without problems, allowing other members of these expanders to be accessed.

Finally, we also provide a mechanism for the programmer to explicitly specify the intended method family, which is useful for handling ambiguous situations. If the receiver in a method invocation has the form `expr with X`, where `expr` is an eJava expression and `X` is an expander name, then only the expander `X` is searched for an appropriate method family; neither the original class of the receiver nor any other expanders are considered. For example, in the scenario described in the last bullet above, the programmer can cause static method-family selection to succeed by explicitly declaring which expander is intended, e.g., `(e with EX2).eval()`. The `with` expression can also be used to multiply expand an expression. For example, the expander `SPX` can be used to adapt an expression `e` of type `Expr` via the syntax `(e with PX) with SPX`, thereby allowing the `safePrint` method to be invoked.

3.2 Dynamic Method Selection

Run-time method lookup in eJava is defined as in Java: the dynamic class of the receiver argument is used to find the most-specific applicable method from the statically determined method family, and that method is then invoked. For example, if the run-time class of `e` in Figure 5 is `Plus`, then the first and third `eval` methods in Figure 4 are applicable, and the latter is the most-specific applicable method. Any local overriding expanders are taken into account when a client invokes a method family in an expander, allowing the new methods to be dispatched to appropriately.

As part of modular static typechecking of each class, Java ensures that each method family is *exhaustive* and *unambiguous*, thereby guaranteeing that run-time method lookup on that family will always succeed. For example, Java checks that a concrete class `C` declares or inherits a concrete method for every method family declared in a superinterface. This check ensures exhaustiveness: instances of `C` and its subclasses are guaranteed to have at least one applicable method in each inherited method family. Java lacks multiple inheritance of classes, so there is no possibility of run-time ambiguities.

eJava includes additional modular static requirements on expanders, in order to continue to ensure that run-time method lookup always succeeds. First, an expander cannot contain an abstract method, even if the class being expanded is abstract. This requirement ensures exhaustiveness in the presence of any unknown concrete subclasses of that class, which must always be assumed to exist given only a modular view of the program. Second, an overriding expander can only expand a class, not an interface. This requirement prevents ambiguities that are not modularly detectable, which can occur if an unknown class implements multiple interfaces. Both requirements are analogous to requirements that exist in prior languages that allow methods to be defined external to their classes [6].

3.3 Preventing Accidental Method Overriding

Our method lookup rules rely on the way in which a program's methods are partitioned into method families. A key property of eJava is the ability to modularly determine the method family to which a method belongs. A method's associated method family is defined to be the same as that of any method that it overrides, and the methods that a particular method overrides can be determined based on information that is available in the static scope where the method is declared. This property of eJava is shared by Java, but it is not shared by other languages that support forms of object adaptation, including AspectJ [17] and Classboxes [4, 3].

To illustrate this distinction, consider again the `Expr` class hierarchy (Figure 1) and the `EX` expander family (Figure 4). As mentioned earlier, if another expander `EX2` for `Expr` also defines an `eval` method, this method (and any `eval` methods in overriding expanders for `EX2`) is considered to belong to a different method family from the `eval` methods for `EX`. If these methods were all considered part of the same method family, there could be run-time ambiguities that are not modularly detectable, for example because

there are two `eval` methods defined for the class `Expr`.

Worse, there may be no ambiguity for a given receiver argument to `eval`, but clients will simply get unexpected behavior at run time. For example, suppose `EX2` defines an `eval` method for some subclass `SpecialPlus` of `Plus`. If both `eval` methods were considered part of the same method family, a client that `uses EX` and invokes `eval` on an instance of `SpecialPlus` would unexpectedly execute the method from `EX2`. Aside from potentially having the wrong behavior, this situation is not modularly typesafe, since for example the `eval` methods in `EX2` could well have a different return type from those in `EX`. Since neither `EX` nor `EX2` is aware of the other modularly, this clash eludes modular typechecking.

Analogous conflicts can occur between a class and an expander, instead of between two expanders. For example, consider a client that uses an expander to class `C` that provides a `print` method, and uses the expander's functionality to display an array of `C`s in an application-specific way. By our semantics, this `print` method is considered to be in its own method family. Therefore, even if the array contains some instances of `D`, a subclass of `C` that provides its own `print` method that does not conform to our application's standards, the client will behave as expected, using only the expander's implementation of `print`. On the other hand, by Classboxes' semantics `D`'s `print` method would be considered part of the same method family as the expander's `print` method, and our application would behave incorrectly. This could not have been prevented by the programmer without requiring global knowledge.

To our knowledge, eJava is the only language for object adaptation that modularly ensures the absence of accidental overriding errors, and hence modularly ensures type safety. Two aspects in AspectJ that define the same method for a given class can easily cause conflicts, and similarly for two classboxes that refine a given class in the same way. These kinds of errors can only be detected with the knowledge of all aspects or classboxes in the program that can affect a given class. For example, in the Classbox/J implementation, clashes involving incompatible return types are only detected by the regular Java typechecker, which runs after the Classbox/J compiler weaves all refinements in the program for a given class into that class's declaration. The next section formalizes our modular type system for eJava and proves its soundness.

4. FEATHERWEIGHT EJAVA

This section describes Featherweight eJava (FeJ), an extension of Featherweight Java (FJ) [16] that formalizes our notion of expanders and its associated modular type system. Aside from making the eJava language semantics precise, FeJ also allows us to prove a type soundness theorem, which validates the sufficiency of our modular type system for ruling out run-time type errors, including problems with accidental overriding. The full details of FeJ and its type soundness proof are available in our companion technical report [28].

4.1 Syntax

The syntax of FeJ is shown in Figure 12. We use notational conventions and sanity conditions analogous to those of FJ. For example, the syntax $\overline{\texttt{D}}$ denotes a sequence of zero or more elements of the domain `D`. Also, an FeJ program consists of a *type table* TT, which maps class, interface, and expander names to their associated declarations, and an expression. We comment on other conventions as necessary throughout this section.

FeJ augments FJ with interfaces and expanders. For simplicity, the `expander` declaration both provides the top expander and all overriding expanders (via the $\overline{\texttt{O}}$ portion of the declaration); local overriding expanders are not modeled. The eJava language largely

```
TD ::= class C extends D implements I̅ {T̅ f̅; K M̅}
     | interface I extends I̅ {MH̅}
     | expander X of T implements I̅ {T̅ f̅=v̅; M̅} O̅
O ::= of C {M̅}
K ::= C(T̅ f̅) {super(f̅); this.f̅=f̅;}
M ::= T m(T̅ x̅) {return t;}
MH ::= T m(T̅ x̅);
T ::= C | I | T^X
t ::= x | t.f | t.m(f̅) | new C(t̅)
    | (T) t | t with X | peel t
v ::= new C(v̅) | v with X
```

Figure 12: The syntax of Featherweight eJava.

$\boxed{\texttt{S<:T}}$

$$\texttt{T<:T}$$

$$\frac{\texttt{S<:T} \qquad \texttt{T<:U}}{\texttt{S<:U}}$$

$$\frac{TT(\texttt{C}) = \texttt{class C extends D implements } \overline{\texttt{I}}\ \texttt{\{...\}}}{\texttt{C<:D}}$$

$$\frac{TT(\texttt{C}) = \texttt{class C extends D implements } \overline{\texttt{I}}\ \texttt{\{...\}}}{\texttt{C<:}\texttt{I}_i}$$

$$\frac{TT(\texttt{I}) = \texttt{interface I extends } \overline{\texttt{J}}\ \texttt{\{...\}}}{\texttt{I<:}\texttt{J}_i}$$

$$\frac{TT(\texttt{X}) = \texttt{expander X of T implements } \overline{\texttt{I}}\ \texttt{\{...\}}\ \overline{\texttt{O}}}{\texttt{T}^{\texttt{X}}\texttt{<:}\texttt{I}_i}$$

$$\frac{\texttt{S<:T}}{\texttt{S}^{\texttt{X}}\texttt{<:T}^{\texttt{X}}}$$

Figure 13: Subtyping in FeJ.

infers where "expansion" and "unexpansion" must occur as part of static method family selection, as described in the previous section. FeJ programs are explicit about expander usage: there are no `use` declarations, an object is expanded via the `with` expression, and an expanded object is "unexpanded" via the `peel` expression. Similarly, we include an explicit type $\texttt{T}^{\texttt{X}}$ for objects of type `T` that are expanded by expander `X`.

4.2 Subtyping

FeJ's subtyping judgment formalizes the relationship between expanded and unexpanded objects; it is shown in Figure 13. The first three rules are from FJ, and the following two rules are natural extensions to handle Java-style interfaces. The next rule is the essence of object adaptation: an expanded object can be typed with any interface that is implemented by the associated expander, allowing the expanded object to be passed wherever values meeting that interface are expected. The final rule extends this ability for object adaptation to instances of any subtype of the type being expanded in an expander declaration.

$\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\frac{\mathit{fields}(\texttt{X}) = \overline{\texttt{T}}\ \overline{\texttt{f}}{=}\overline{\texttt{v}}}{\texttt{(v with X).}\texttt{f}_i \longrightarrow \texttt{v}_i}$$

$$\frac{\mathit{fields}(\texttt{X}) = \overline{\texttt{T}}\ \overline{\texttt{g}}{=}\overline{\texttt{v}} \qquad \texttt{f} \notin \overline{\texttt{g}}}{\texttt{(v with X).f} \longrightarrow \texttt{v.f}}$$

$$\frac{\texttt{v} = \texttt{new C}(\overline{\texttt{v}}) \qquad \mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t}_0)}{\texttt{(v with X).m}(\overline{\texttt{u}}) \longrightarrow [\overline{\texttt{x}} \mapsto \overline{\texttt{u}}, \texttt{this} \mapsto \texttt{v with X}]\texttt{t}_0}$$

$$\frac{\texttt{v} = \texttt{v}'\ \texttt{with X}' \qquad \mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{Object}) = (\overline{\texttt{x}}, \texttt{t}_0)}{\texttt{(v with X).m}(\overline{\texttt{u}}) \longrightarrow [\overline{\texttt{x}} \mapsto \overline{\texttt{u}}, \texttt{this} \mapsto \texttt{v with X}]\texttt{t}_0}$$

$$\frac{\begin{array}{c}\mathit{TT}(\texttt{X}) = \texttt{expander X of T implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}{=}\overline{\texttt{v}}\texttt{;}\ \overline{\texttt{M}}\}\ \overline{\texttt{O}} \\ \texttt{m is not defined in } \overline{\texttt{M}}\end{array}}{\texttt{(v with X).m}(\overline{\texttt{u}}) \longrightarrow \texttt{v.m}(\overline{\texttt{u}})}$$

$$\texttt{peel}\ (\texttt{v with X}) \longrightarrow \texttt{v}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{t}_0\ \texttt{with X} \longrightarrow \texttt{t}_0'\ \texttt{with X}}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{peel t}_0 \longrightarrow \texttt{peel t}_0'}$$

Figure 14: FeJ evaluation rules.

$\boxed{\mathit{fields}(\texttt{X}) = \overline{\texttt{T}}\ \overline{\texttt{f}}\ \texttt{=}\ \overline{\texttt{v}}}$

$$\frac{\mathit{TT}(\texttt{X}) = \texttt{expander X of T implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}\ \texttt{=}\ \overline{\texttt{v}}\texttt{;}\ \overline{\texttt{M}}\}\ \overline{\texttt{O}}}{\mathit{fields}(\texttt{X}) = \overline{\texttt{T}}\ \overline{\texttt{f}}{=}\overline{\texttt{v}}}$$

$\boxed{\mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})}$

$$\frac{\begin{array}{c}\mathit{TT}(\texttt{X}) = \texttt{expander X of T implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}{=}\overline{\texttt{v}}\texttt{;}\ \overline{\texttt{M}}\}\ \overline{\texttt{O}} \\ \texttt{of C } \{\overline{\texttt{M}'}\} \in \overline{\texttt{O}} \qquad \texttt{U m}(\overline{\texttt{U}}\ \overline{\texttt{x}})\ \{\texttt{return t;}\} \in \overline{\texttt{M}'}\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})}$$

$$\frac{\begin{array}{c}\mathit{TT}(\texttt{X}) = \texttt{expander X of T implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}{=}\overline{\texttt{v}}\texttt{;}\ \overline{\texttt{M}}\}\ \overline{\texttt{O}} \\ \texttt{of C } \{\overline{\texttt{M}'}\} \in \overline{\texttt{O}} \qquad \texttt{m is not defined in } \overline{\texttt{M}'} \\ \mathit{TT}(\texttt{C}) = \texttt{class C extends D} \cdots\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{D})}$$

$$\frac{\begin{array}{c}\mathit{TT}(\texttt{X}) = \texttt{expander X of T implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}{=}\overline{\texttt{v}}\texttt{;}\ \overline{\texttt{M}}\}\ \overline{\texttt{O}} \\ \texttt{C is not defined in } \overline{\texttt{O}} \\ \mathit{TT}(\texttt{C}) = \texttt{class C extends D} \cdots\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{D})}$$

$$\frac{\begin{array}{c}\mathit{TT}(\texttt{X}) = \texttt{expander X of T implements } \overline{\texttt{I}}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}{=}\overline{\texttt{v}}\texttt{;}\ \overline{\texttt{M}}\}\ \overline{\texttt{O}} \\ \texttt{U m}(\overline{\texttt{U}}\ \overline{\texttt{x}})\ \{\texttt{return t;}\} \in \overline{\texttt{M}}\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{X}, \texttt{Object}) = (\overline{\texttt{x}}, \texttt{t})}$$

Figure 15: Helper rules for FeJ evaluation.

A notable absence from the subtyping relation is the axiom $\texttt{T}^{\texttt{X}}\texttt{<:T}$. Omitting this rule forces an expanded object to be unexpanded via `peel` before it is passed where a value of the original type is expected. It also ensures that methods in an expander will not be treated as overriding methods in the unexpanded type. Both of these behaviors mirror the semantics and implementation strategy of the eJava language. Despite the absence of this subtyping relationship, values of type $\texttt{T}^{\texttt{X}}$ may still access methods and fields of the original type `T`, as we show below. In this way, for example, an expanded type may "inherit" methods from the original type in order to meet a new interface.

4.3 Dynamic Semantics

Figure 14 provides the small-step operational semantics of FeJ and makes use of the helper rules in Figure 15. We only present the rules that relate to expanders; the rest of FeJ is identical to FJ. The first two rules define field lookup for expanded objects. If the expander defines the field being accessed, then its associated value is returned. Otherwise, field lookup proceeds in the unexpanded object.

The rules for method lookup on expanded objects are described next, and they depend on the *mbody* helper function, which finds the best implementation of the method `m` for class `C` in expander `X`. There are four cases. If `X` has an overriding expander for `C` containing a method `m`, then that method's body is returned. If `X` either has an overriding expander for `C` but that expander does not override `m`, or if `X` has no overriding expander for `C`, then we recursively search for an overriding expander for `C`'s direct superclass `D`. Finally, if `C` = `Object`, then recursion ends and the body of the `m` method in the top expander is returned. Although `Object` may not be a subtype of the type of the expander, static typechecking ensures that *mbody* is always used in a type-correct manner.

Given this *mbody* function, the third rule in Figure 14 looks up the appropriate method body in `X` for the unexpanded object's run-time class `C`. The next rule handles the case when the expanded value itself has the form $\texttt{v}'\ \texttt{with X}'$. In that case, the top expander's method is always invoked, without considering overriding expanders. This semantics makes sense since the run-time type of the expanded value has the form $\texttt{T}^{\texttt{X}'}$, and such a type cannot be a subtype of any class type according to our subtype relation, so we are guaranteed that no overriding expanders are applicable. The semantics is encoded by invoking *mbody* with `Object` as the third argument. Finally, if the expander `X` does not define `m`, then lookup proceeds in the unexpanded object.

The rest of the rules in Figure 14 define the semantics of `with` and `peel`, which are straightforward.

4.4 Static Semantics

The rules for typechecking terms are presented in Figure 16, and helper functions are defined in Figure 17. Again, only the rules related to expanders are shown. A field access is type-correct if an appropriate field can be found for the type of the receiver. For a receiver of type $\texttt{U}^{\texttt{X}}$, the declaration of `X` is searched for a field of the appropriate name. If the field is not found, the type `U` is searched for a definition of the field. In this way, expanded objects are allowed to access fields of the unexpanded object (if they are not shadowed by the expander). Typechecking of method invocation proceeds analogously. The type rule for an expression `t with X` ensures that the type of `t` is a subtype of the type expanded by `X`. The type rule for `peel` is straightforward.

Finally, Figure 18 contains the rules for typechecking expander

$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \mathit{ftype}(\mathtt{f}, \mathtt{T}) = \mathtt{U}}{\Gamma \vdash \mathtt{t.f} : \mathtt{U}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_0 \\ \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} \\ \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{T}}\end{array}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{T}}$$

$$\frac{\begin{array}{c}TT(\mathtt{X}) = \texttt{expander X of T implements } \overline{\mathtt{I}} \ \{\ldots\} \ \overline{\mathtt{O}} \\ \Gamma \vdash \mathtt{t} : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{t} \texttt{ with } \mathtt{X} : \mathtt{U}^{\mathtt{X}}}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}^{\mathtt{X}}}{\Gamma \vdash \texttt{peel } \mathtt{t} : \mathtt{T}}$$

Figure 16: FeJ typechecking for terms.

$\boxed{\mathit{ftype}(\mathtt{f}, \mathtt{T}) = \mathtt{U}}$

$$\frac{\mathit{fields}(\mathtt{X}) = \overline{\mathtt{T}} \ \overline{\mathtt{f}}{=}\overline{\mathtt{v}}}{\mathit{ftype}(\mathtt{f}_i, \mathtt{U}^{\mathtt{X}}) = \mathtt{T}_i}$$

$$\frac{\mathit{fields}(\mathtt{X}) = \overline{\mathtt{T}} \ \overline{\mathtt{g}}{=}\overline{\mathtt{v}} \qquad \mathtt{f}_i \notin \overline{\mathtt{g}}}{\mathit{ftype}(\mathtt{f}_i, \mathtt{U}^{\mathtt{X}}) = \mathit{ftype}(\mathtt{f}_i, \mathtt{U})}$$

$\boxed{\mathit{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{T}} \to \mathtt{T}}$

$$\frac{\begin{array}{c}TT(\mathtt{X}) = \texttt{expander X of T implements } \overline{\mathtt{I}} \ \{\overline{\mathtt{T}} \ \overline{\mathtt{f}}{=}\overline{\mathtt{v}}; \ \overline{\mathtt{M}}\} \ \overline{\mathtt{O}} \\ \mathtt{U} \ \mathtt{m}(\overline{\mathtt{U}} \ \overline{\mathtt{x}}) \ \{\texttt{return t;}\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{S}^{\mathtt{X}}) = \overline{\mathtt{U}} \to \mathtt{U}}$$

$$\frac{\begin{array}{c}TT(\mathtt{X}) = \texttt{expander X of T implements } \overline{\mathtt{I}} \ \{\overline{\mathtt{T}} \ \overline{\mathtt{f}}{=}\overline{\mathtt{v}}; \ \overline{\mathtt{M}}\} \ \overline{\mathtt{O}} \\ \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{S}^{\mathtt{X}}) = \mathit{mtype}(\mathtt{m}, \mathtt{S})}$$

Figure 17: Helper rules for term typechecking.

$\boxed{\mathtt{TD} \text{ OK}}$

$$\frac{\begin{array}{c}\bullet \vdash \overline{\mathtt{v}} : \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{T}} \\ \overline{\mathtt{M}} \text{ OK in } \mathtt{X}, \mathtt{T} \qquad \overline{\mathtt{O}} \text{ OK in } \mathtt{X} \\ \mathit{reallyImplements}(\mathtt{T}^{\mathtt{X}}, \overline{\mathtt{I}})\end{array}}{\texttt{expander X of T implements } \overline{\mathtt{I}} \ \{\overline{\mathtt{T}} \ \overline{\mathtt{f}}{=}\overline{\mathtt{v}}; \ \overline{\mathtt{M}}\} \ \overline{\mathtt{O}} \text{ OK}}$$

$\boxed{\mathtt{O} \text{ OK in } \mathtt{X}}$

$$\frac{\begin{array}{c}TT(\mathtt{X}) = \texttt{expander X of T implements } \overline{\mathtt{I}} \ \{\ldots\} \ \overline{\mathtt{O}} \\ \mathtt{C} <: \mathtt{T} \qquad \overline{\mathtt{M}} \text{ OverrideOK in } \mathtt{X}, \mathtt{C}\end{array}}{\texttt{of C } \{\overline{\mathtt{M}}\} \text{OK in X}}$$

$\boxed{\mathtt{M} \text{ OK in } \mathtt{X}, \mathtt{T}}$

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{T}^{\mathtt{X}} \vdash \mathtt{t}_0 : \mathtt{U}_0 \qquad \mathtt{U}_0 <: \mathtt{T}_0}{\mathtt{T}_0 \ \mathtt{m}(\overline{\mathtt{T}} \ \overline{\mathtt{x}}) \ \{\texttt{return } \mathtt{t}_0;\} \text{ OK in } \mathtt{X}, \mathtt{T}}$$

$\boxed{\mathtt{M} \text{ OverrideOK in } \mathtt{X}, \mathtt{C}}$

$$\frac{\begin{array}{c}\mathit{override}(\mathtt{m}, \mathtt{X}, \overline{\mathtt{T}} \to \mathtt{T}_0) \\ \mathtt{T}_0 \ \mathtt{m}(\overline{\mathtt{T}} \ \overline{\mathtt{x}}) \ \{\texttt{return } \mathtt{t}_0;\} \text{ OK in } \mathtt{X}, \mathtt{C}\end{array}}{\mathtt{T}_0 \ \mathtt{m}(\overline{\mathtt{T}} \ \overline{\mathtt{x}}) \ \{\texttt{return } \mathtt{t}_0;\} \text{ OverrideOK in } \mathtt{X}, \mathtt{C}}$$

$\boxed{\mathit{reallyImplements}(\mathtt{T}, \mathtt{I})}$

$$\frac{\begin{array}{c}TT(\mathtt{I}) = \texttt{interface I extends } \overline{\mathtt{J}} \ \{\overline{\mathtt{MH}}\} \\ \mathtt{S} \ \mathtt{m}(\overline{\mathtt{S}} \ \overline{\mathtt{x}}) \texttt{;} \in \overline{\mathtt{MH}} \text{ implies } \mathit{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{U}} \to \mathtt{U} \text{ and } \mathit{override}(\mathtt{m}, \mathtt{I}, \overline{\mathtt{U}} \to \mathtt{U}) \\ \mathit{reallyImplements}(\mathtt{T}, \overline{\mathtt{J}})\end{array}}{\mathit{reallyImplements}(\mathtt{T}, \mathtt{I})}$$

$\boxed{\mathit{override}(\mathtt{m}, \mathtt{X}, \overline{\mathtt{T}} \to \mathtt{T}_0)}$

$$\frac{\begin{array}{c}TT(\mathtt{X}) = \texttt{expander X of T implements } \overline{\mathtt{I}} \ \{\overline{\mathtt{T}} \ \overline{\mathtt{f}}{=}\overline{\mathtt{v}}; \ \overline{\mathtt{M}}\} \ \overline{\mathtt{O}} \\ \mathtt{U} \ \mathtt{m}(\overline{\mathtt{U}} \ \overline{\mathtt{x}}) \ \{\texttt{return t;}\} \in \overline{\mathtt{M}}\end{array}}{\mathit{override}(\mathtt{m}, \mathtt{X}, \overline{\mathtt{U}} \to \mathtt{U})}$$

Figure 18: Typechecking for expander declarations.

declarations. We assume that the type being expanded in a top expander is distinct from each of the classes expanded in overriding expanders, and that these classes are distinct from one another. Each overriding expander is required to be for a class that is a subtype of the type expanded by the top expander. The third rule describes how methods are typechecked. The method body is typechecked under the assumption that the receiver `this` has the expanded type. The OverrideOK judgment is used for typechecking methods in overriding expanders. In addition to the ordinary rules for typechecking methods, this judgment requires that a method of the same type signature appear in the top expander. That check is accomplished via the *override* helper function. Finally, the *reallyImplements* function ensures that each expander truly meets its declared interfaces.

A key property of this type system is *modularity*. Each class, interface, and expander is typechecked using only knowledge of its own declaration and the declaration of other declarations that it directly references. This means, for example, that an expander is typechecked without knowledge of what other expanders exist for the type being expanded, and without knowledge of all subtypes of the type being expanded.

4.5 Type Soundness

We have proven a type soundness theorem for FeJ using the standard "progress and preservation" style [29].

THEOREM 4.1. *(Progress) If* $\bullet \vdash \mathtt{t} : \mathtt{T}$*, then either* `t` *is a value,* `t` *contains a subexpression of the form* `(U)(v)` *where* $\bullet \vdash \mathtt{v} : \mathtt{S}$ *and* $\mathtt{S} \not<: \mathtt{U}$*, or there exists some term* `s` *such that* $\mathtt{t} \longrightarrow \mathtt{s}$.

THEOREM 4.2. *(Type Preservation) If* $\Gamma \vdash \mathtt{t} : \mathtt{T}$ *and* $\mathtt{t} \longrightarrow \mathtt{s}$*, then there exists some type* `S` *such that* $\Gamma \vdash \mathtt{s} : \mathtt{S}$ *and* $\mathtt{S} <: \mathtt{T}$.

The full proofs of these theorems are available in our companion technical report [28]. Together the theorems imply that well-typed FeJ programs cannot incur a type error at run time. This means that FeJ's modular type system is sufficient to guarantee the absence of the kinds of type errors that are caused by accidental overriding, as described in Section 3.3.

5. COMPILATION

The eJava compiler is built on top of the Polyglot extensible compiler framework for Java [20]. Like other Polyglot-based language implementations, our compiler translates eJava programs into equivalent Java programs, which can then be compiled with a standard Java compiler and run on a standard Java virtual machine. Polyglot compiles Java 1.4, so our eJava compiler inherits this limitation, for example lacking support for generics. We expect the issues for generic expanders to be analogous to those for classes, and we plan to pursue such an extension in the future.

The eJava compiler translates each expander family to a single Java class with the same name. Figure 19 shows the code generated for the `EX` expander from Figure 4. We refer to that example throughout this section.

5.1 Wrapper Classes

The functionality provided by an expander family is implemented in a set of *wrapper classes*, one for each expander in the family. In our example, the wrapper classes `EX$Expr`, `EX$Value`, and `EX$Plus` are generated for the expanders of `EX` whose associated classes are `Expr`, `Value`, and `Plus`, respectively.

These classes are called *wrappers* because they hold a reference to the original (unexpanded) object. This reference is used in the implementation of the methods of the expander. For instance, `EX$Plus`'s `eval` method uses it to call the `op1` and `op2` methods of its associated object. Note that the hierarchy of the wrapper classes mimics the hierarchy of the types for which they were created. Consequently, in any given family, expanders of more specific types inherit—and may override, when desired—methods from expanders of less specific types.

5.2 The `expand` Method

Wrapper objects are created by the expander's `expand` method. `expand` takes an instance of the top expander's associated type as its argument and performs `instanceof` tests to determine the best (i.e., most specific) wrapper class to use for that particular object. The first part of the `expand` method in Figure 19 is used to properly handle fields in expanders; it is discussed in Section 5.4.

Clients of an expander call its `expand` method in order to access functionality it provides. For example, in Figure 5 the call `e.eval()` is translated to `(EX.expand(e)).eval()`.

5.3 Interface-Implementing Expanders

Expander families that implement one or more interfaces, such as Figure 6's `PX`, which adds the `Printable` interface to instances of `Expr`, are translated as described above. Additionally, the wrapper class generated for the top expander's associated type is declared to implement those interfaces. This enables the client to use the expanded object (returned by `expand`) in contexts where one of the interface types is expected.

An expander can employ methods of the type being expanded in order to meet new interfaces. To implement this functionality, we generate forwarding methods in the wrapper class for the top expander, which simply invoke the corresponding methods on the unexpanded object. All interfaces in Java implicitly include signatures for the public methods of `java.lang.Object`, like `equals`, `toString`, and `hashCode`, so forwarding methods for them are also created (except in cases where the method is explicitly shadowed by the expander).

These forwarding methods are only ever used when an expanded object is viewed through one of the interfaces that the expander meets. They are not necessary in all other situations, because in those cases eJava's semantics will cause the appropriate methods to be called directly on the unexpanded method. For example, if `e` has static type `Expr` then `e.toString()` will always invoke the `toString()` method defined for the run-time class of `e`, without considering any of the expanders in scope.

When an expanded object is viewed through one of the interfaces that the expander meets, the forwarding methods enable the expanded object to "pretend" to be the original object as much as possible. However, one well-known limitation of wrappers is that they do not preserve object identity. Therefore, code that employs `==` or `instanceof` on a value of some interface type can get incorrect results if passed an expanded object. This limitation is an artifact of our modular compilation strategy and is also a limitation of the adapter design pattern [14].

An alternative compilation strategy that nonmodularly modifies a class in place would not suffer from this problem, but it would suffer from other problems. For example, it would be impossible to adapt a class to support two incompatible interfaces (e.g., interfaces that define the same method but with a different return type) in two different expanders. Both the AspectJ and Classboxes implementation strategies suffer from these kinds of problems.

5.4 Handling State in Expanders

One straightforward way to represent the fields of an expander family in our implementation scheme would be to declare those

```
class EX {
  public static final java.util.WeakHashMap<Expr, EX$$tate> cache = new java.util.WeakHashMap<Expr, EX$$tate>();
  public static class EX$$tate {
    public java.lang.ref.WeakReference<Expr> instance$ = null;
    public java.lang.ref.WeakReference<EX$Expr> wrapper$ = null;
  }
  public static class EX$Expr {
    protected final Expr instance;
    protected final EX$$tate state;
    protected EX$Expr(final Expr instance, EX$$tate state) {
      this.instance = instance;
      this.state = state;
    }
    public Value eval() {
      throw new EvalError();
    }
  }
  public static class EX$Plus extends EX$Expr {
    protected EX$Plus(final Expr instance, EX$$tate state) {
      super(instance, state);
    }
    public Value eval() {
      Value v1 = (EX.expand(instance.op1())).eval();
      Value v2 = (EX.expand(instance.op2())).eval();
      // ...
    }
  }
  public static class EX$Value extends EX$Expr {
    protected EX$Value(final Expr instance, EX$$tate state) {
      super(instance, state);
    }
    public Value eval() {
      return (Value) instance;
    }
  }
  public static synchronized EX$Expr expand(final Expr instance) {
    EX$Expr r;
    EX$$tate state = cache.get(instance);
    if (state == null) {
      state = new EX$$tate();
      state.instance$ = new java.lang.ref.WeakReference<Expr>(instance);
      cache.put(instance, state);
    }
    else if (state.wrapper$ != null) {
      r = state.wrapper$.get();
      if (r != null)
        return r;
    }
    if (instance instanceof Plus)
      r = new EX$Plus(instance, state);
    else if (instance instanceof Value)
      r = new EX$Value(instance, state);
    else
      r = new EX$Expr(instance, state);
    state.wrapper$ = new java.lang.ref.WeakReference<EX$Expr>(r);
    return r;
  }
}
```

Figure 19: Java translation of the `EX` expander from Figure 4.

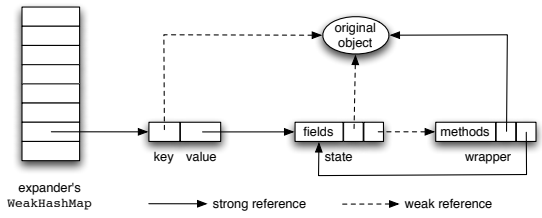

Figure 20: Expander state and wrapper objects.

fields in the wrapper class generated for that family's top expander. Wrapper objects would additionally be cached for use by the `expand` method, in order to give clients of the same object a consistent view of its new fields' values. Unfortunately, this simple approach interacts poorly with Java's garbage collection: `expand`'s cache would hold a reference to every object ever adapted by the expander, preventing those objects from ever being garbage collected.

Java has a notion of *weak references*, which, unlike the usual *strong references*, do not prevent their referents from being collected. One possible solution to the above problem is for the cache to hold weak references to unexpanded objects. Although this approach does not result in memory leaks, it suffers from the even more serious "disappearing object problem": it is possible for the original object to be garbage collected while one or more clients still hold a reference to the associated wrapper object.

Our implementation scheme, outlined in Figure 20, avoids both of these problems. The eJava compiler generates a *state class* for each expander, and each wrapper object maintains a reference to an associated state object. The `EX$$state` class in Figure 19 is the state class generated for the `EX` expander. If `EX` declared a field `f`, it would be declared in this state class. Client code referring to `e.f`, where `e` has type `Expr`, would be translated to `EX.expand(e).state.f`.

Like wrappers, state objects are instantiated when an object is expanded. As in the approach discussed above, the `expand` method uses a `WeakHashMap`, a hashtable implementation that holds weak references to the objects it uses as keys, to cache the state objects for each object expanded. An important property of `WeakHashMap` is that values associated with keys that are garbage collected are automatically removed. This feature ensures that state objects whose associated unexpanded objects are no longer accessible are properly disposed of, avoiding memory leaks.

For performance reasons, the state object contains a `wrapper$` field, which is used to cache the associated wrapper object. This allows the implementation to avoid instantiating a new wrapper object every time a client accesses functionality by an expander on the same object.

To solve the disappearing object problem, the wrappers generated by our translation hold strong references to their original objects. This ensures that the original object will not be garbage collected as long as there is at least one client who still holds a reference to the wrapper.

Should it ever be the case that no client holds a reference to the original object or to its wrapper, the wrapper's strong reference to the object will not keep the object from being garbage collected. In such scenarios, the only object in the system which has a reference to the wrapper is the state object. And because this reference is weak, it does not prevent the wrapper—and consequently the original object—from being collected.

Finally, suppose one or more clients have strong references to the original object, but no client has a strong references to the associated wrapper. Here, it is possible that the wrapper will be garbage collected. Fortunately, because our wrappers are stateless, they can be instantiated as often as needed. This is done in the expander's `expand` method using the state object's `instance$` field (see Figure 19).

```
class FancyCalculator$EX extends EX {
  public static class EX$Minus extends EX$Expr {...}
  public static EX$Expr expand(final Expr instance) {
    EX$Expr r = null;
    EX$$state state = cache.get(instance);
    if (state == null) {
      state = new EX$$state();
      state.instance$ =
        new java.lang.ref.WeakReference<Expr>(instance);
      cache.put(instance, state);
    }
    else if (state.wrapper$ != null) {
      r = state.wrapper$.get();
      if (r != null &&
          r.creator$ == FancyCalculator$EX.class)
        return r;
    }
    if (instance instanceof Minus)
      r = new EX$Minus(instance, state);
    else
      return EX.expand(instance);
    r.creator$ = FancyCalculator$EX.class;
    state.wrapper$ =
      new java.lang.ref.WeakReference<EX$Expr>(r);
    return r;
  }
}
```

Figure 21: The translation of the local overriding expander in Figure 11.

5.5 The `expand` Method, Revisited

Now that most of eJava's implementation details have been revealed, we are able to provide a more detailed description of the actions carried out by the `expand` method.

Step 1: `expand` searches the expander's cache for the state object associated with the object to be expanded. If the search succeeds *and* the state object's wrapper exists, `expand` returns the wrapper object. If the search fails, a new state object whose `instance$` field references the instance to be expanded is created and stored in the cache.

Step 2: If we get to this point, we've got a state object, but no wrapper. `expand` then creates the most specific wrapper based on the dynamic type of the object to be expanded. A weak reference to this object is stored in the state object, and the wrapper is returned to the caller.

5.6 Local Overriding Expanders

As discussed in section 2.5, clients may locally augment an expander family with any number of local overriding expanders. Our compilation scheme supports this by generating a subclass of the original expander's implementation class. For example, Figure 21 shows the code generated for the local overriding expander shown

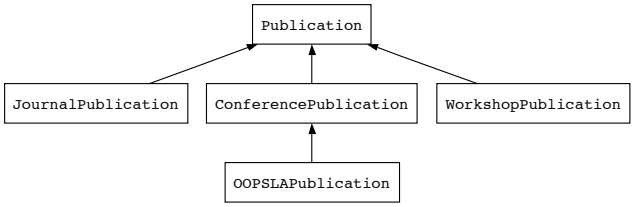

Figure 22: The hierarchy of publication classes.

in Figure 11. This class contains the appropriate wrapper classes, which are declared to be subclasses of the wrapper classes of the original expander. The state class created for the original expander is inherited for use by the local overriding expander. When creating new wrapper objects, the new expander class's `expand` method uses `instanceof` tests to determine whether one of its refinements should be used for the instance being expanded. If this is not the case, it forwards the call to the `expand` method of its superclass. The uses of locally augmented expanders are compiled as calls to the new expander class's `expand` method (and not that of the original expander).

To be *hygienic*, as discussed in Section 2.5, wrappers created by local expanders should never be returned by `expand` in contexts other than their own. Similarly, a wrapper created for a particular instance by the original expander should not be used in a context where a local overriding expander exists for the dynamic type of that instance. In order to prevent these scenarios, an expander's `expand` method must never return a cached wrapper object that it did not create itself. For example, Figure 21 shows the fragment of `FancyCalculator$EX`'s `expand` method that enforces this restriction. Since any expander family may be augmented by clients through local overriding expanders, this check is performed by the `expand` method of every expander, regardless of whether or not the the existence of local overriding expanders is known at compile time (for clarity, this mechanism was omitted from the implementation of `EX` shown in Figure 19).

6. EXPERIENCE

This section describes two experiments we performed to gauge the practical utility of eJava's expanders. First, we considered a scenario in which a client must adapt existing objects in order to display them in a tree structure using Swing's `JTree` class. We implemented solutions in both Java and eJava to compare the benefits and limitations of each approach. Second, we performed an exploratory study of the Eclipse IDE framework [7]. The goal of the study was to understand the various extensibility idioms used in Eclipse and the extent to which expanders allow these idioms to be expressed more naturally and/or reliably.

6.1 Making Objects "Swing"

Consider a database of objects that represent various academic publications, with each publication linking to the publications it references. The publications in this database are represented by the `Publication` hierarchy, shown in Figure 22. Since different clients of this database are likely to want to display publications in different ways, the classes in the `Publication` hierarchy do not provide any displaying capabilities on their own; this functionality must be implemented by each client that requires it. Further, because the publication objects come from a database (and are not instantiated by clients themselves), this functionality must be added *externally*.

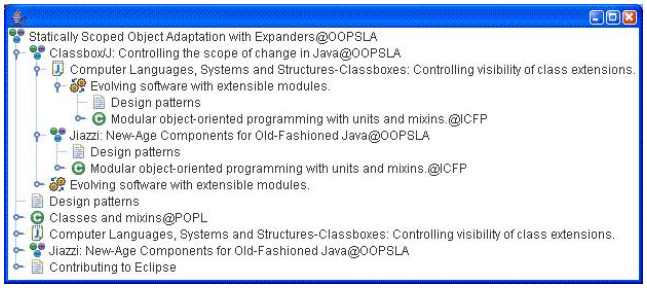

Figure 23: The client application.

```
public interface ILabelProvider {
  String getText();
  Icon getIcon();
}
```

Figure 24: The `ILabelProvider` interface.

This case study addresses how one client might display the objects in this database graphically as a tree—with each node representing a publication and subnodes representing the publication's references—using Swing's `JTree` class. Figure 23 shows a screenshot of the client application. Note that each kind of publication is displayed in its own style, consisting of a particular icon and text format.

To display objects in a `JTree`, clients must provide an implementation of Java's `TreeModel` interface, which provides a data model for tree. As a convenience, the Swing library provides a default implementation of this interface, `DefaultTreeModel`. However, `DefaultTreeModel` requires the objects in the tree to implement the `TreeNode` interface. Therefore, in order to avoid duplicating the code of `DefaultTreeModel`, the publication objects must be adapted to meet the `TreeNode` interface.

To customize the way objects in a `JTree` are displayed, the client must also implement the `TreeCellRenderer` interface. This interface has a single method, `getTreeCellRendererComponent`, which takes an `Object` and returns the graphical `Component` to be used for displaying that object. The simplicity of this interface often complicates the implementation of the `getTreeCellRendererComponent` method, which must perform `instanceof` tests and typecasts if different kinds of objects should be displayed differently. Therefore, many Swing applications define their own interface for rendering, which is implemented by the object passed to `getTreeCellRendererComponent`. The implementation of that method can then delegate most of the rendering work to the given object by calling appropriate methods from this new interface. In our application, we require the objects in the tree to support the `ILabelProvider` interface, shown in Figure 24, which allows each kind of publication to provide its own text and icon for display. Therefore, `Publication` objects must also be adapted to meet the `ILabelProvider` interface.

6.1.1 The eJava Version

Our eJava implementation uses expanders to adapt the various types of publications to meet the `TreeNode` and `ILabelProvider` interfaces. Figure 25 shows some of the code in the top expander (i.e., the one for `Publication`). The first several methods in the expander implement the `TreeNode` interface. Please note the

```
use StringExp;
public expander PublicationExp of Publication
     implements TreeNode, ILabelProvider {
  private Vector children = null;
  public Enumeration children() {
    if (children == null) {
      children = new Vector();
      Iterator it = citations().iterator();
      while (iterator.hasNext()) {
        Publication p = (Publication)it.next();
        children.add(p with PublicationExp);
      }
    }
    return children.elements();
  }
  // ...
  public Icon getIcon() {
    return "/icons/publication.gif".getIcon();
  }
  public String getText() {
    return getTitle();
  }
}
```

Figure 25: The `PublicationExp` expander.

```
expander PublicationExp of ConferencePublication {
  public Icon getIcon() {
    return "/icons/conference.gif".getIcon();
  }
  public String getText() {
    return getTitle() + "@" + getConferenceName();
  }
}
```

Figure 26: An overriding expander.

use of the `with` operator to explicitly expand `p` (an instance of `Publication`) in the `children` method. This is necessary because we intend to store the expanded version of those objects in a `Vector`. Since `Vector`'s `add` method takes an argument of type `Object`, no adaptation is necessary, and therefore `p` is not expanded automatically. Java 1.5's generics would allow us to type `children` as `Vector<TreeNode>`, which would then result in the implicit adaptation of the object, but as mentioned earlier, the eJava compiler does not currently support generics.

The expander in Figure 25 provides a default implementation of `ILabelProvider` for instances of `Publication`. Overriding expanders customize the presentation of each kind of publication by providing their own implementation of the `getText` and `getIcon` methods. For example, Figure 26 shows the overriding expander for `ConferencePublication`.

We use an expander for `String` to encapsulate the logic for loading icons from files, as shown in Figure 27. The expander also caches the icon in its `icon` field. Since in Java identical string literals are represented by the same underlying `String` object, our application will not load an icon more than once from the same file.

6.1.2 The Java Version

```
public expander StringExp of String {
  private Icon icon = null;
  public Icon getIcon() {
    if (icon == null)
      icon =
        new ImageIcon(Object.class.getResource(this));
    return icon;
  }
}
```

Figure 27: Retrieving icons from a file.

```
public class ConferenceAdapter
     extends PublicationAdapter
  implements ILabelProvider {
  public ConferenceAdapter(ConferencePublication p) {
    super(p);
  }
  public Icon getIcon() {
    return IconCache.getIcon("/icons/conference.gif");
  }
  public String getText() {
    ConferencePublication a =
      (ConferencePublication) adaptee;
    return a.getTitle() + "@" + a.getConferenceName();
  }
}
```

Figure 28: An adapter for `ConferencePublication`.

Our Java implementation employs the adapter design pattern [14]. To encode special rendering behavior for each subclass of `Publication`, we create a hierarchy of adapter classes that mirrors the publication hierarchy. The root of the adapter hierarchy is `PublicationAdapter`, which looks similar to the `PublicationExp` expander, except that it maintains an explicit field `adaptee` of type `Publication` to access the underlying publication object. Similarly, the other classes in the hierarchy of adapters correspond to the overriding expanders of the eJava implementation. For example, the `ConferenceAdapter`, which is the analogue of the overriding expander for `ConferencePublication` (Figure 26) is shown in Figure 28. Since the adapter makes use of specific features of `ConferencePublication`, like `getConferenceName`, the `adaptee` field must be downcast appropriately.

Our Java implementation is required to perform `instanceof` tests in order to determine the correct adapter class for a publication object. The `createAdapter` method in Figure 29 implements this functionality. The `instanceof` tests must be performed in most- to least-specific order to ensure that each object is adapted properly. It is possible to do away with the adapter hierarchy and instead use a single adapter class for all publications. However, in that case this sequence of `instanceof` tests would be required in the implementation of every method that requires specialized behavior for `Publication` subclasses, instead of occurring only once at the point of adaptation.

6.1.3 Comparison

This case study exposes several advantages of expanders over standard adapters in Java. First, the eJava version is more eas-

```
public class GUIAdapterFactory {
  public static PublicationAdapter
    createAdapter(Publication p) {
    if (p instanceof OOPSLAPublication) {
      OOPSLAPublication pub = (OOPSLAPublication) p;
      return new OOPSLAAdapter(pub);
    }
    if (p instanceof ConferencePublication) {
      ConferencePublication pub =
        (ConferencePublication) p;
      return new ConferenceAdapter(pub);
    }
    //....
    return new PublicationAdapter(pub);
  }
}
```

Figure 29: The adapter factory.

ily extensible. For example, consider the task of adding special support for `TOPLASPublication`, which is a subclass of `JournalPublication`. In the eJava version, one would simply add a new overriding expander for `PublicationExp`, either in the same source file as the original expanders or via local expander overriding. In the Java version, one would analogously need to add a new adapter class, `TOPLASAdapter`, which inherits from `JournalAdapter`. However, it would also be necessary to update or create a subclass of the `GUIAdapterFactory` (from Figure 29), in order to add a new case to `createAdapter`'s cascading `if` statement that appropriately adapts a `TOPLASPublication`. Further, care must be taken to place this case in the appropriate order with respect to the existing cases.

The eJava version is also less error prone than the Java version. The latter version requires `instanceof` tests and associated downcasts in `GUIAdapterFactory`, in order to create the adapters, and additional downcasts within the adapters to narrow the type of the `adaptee` field, as shown earlier. In contrast, the eJava version requires none of these `instanceof` tests or downcasts. Instead, eJava's modular type system ensures type correctness statically. In total, the Java version of our simple application requires four `instanceof` tests and eight downcasts. The eJava version contains no `instanceof` tests and three downcasts. Two of these downcasts would have been avoided with the use of generics, and the third occurs in the implementation of `TreeCellRenderer`, in order to narrow the `Object` argument of `getTreeCellRendererComponent` to `ILabelProvider`; it is also present in the Java version.

Finally, the eJava version has a performance advantage over the Java version. As described in Section 5, the instances of wrapper classes used in the implementation of an expander are cached for each expanded object. Therefore, the tree display in our application will create exactly one wrapper object per publication object displayed, even if the publication appears multiple times in the tree. In the Java version, each node in the tree has a separate adapter instance. Similarly, we remarked earlier that the `StringExp` expander naturally ensures that each icon is loaded from a file at most once. In the Java version, this optimization had to be implemented with an explicit icon cache.

6.2 An Exploratory Study of Eclipse

Eclipse is built with extensibility as a primary goal: the framework is structured as a small kernel, known as the Eclipse *platform*, and a collection of external *plugins* that provide the bulk of the system's functionality. As such, Eclipse represents a rich source for understanding the state of the art in Java-based extensibility idioms. In the rest of this section, we describe several of these idioms and their relationship to eJava's expanders.

6.2.1 External Methods

Eclipse uses two main approaches to add methods to existing classes from the outside. First, it provides two kinds of visitors: `ASTVisitor`, for traversing the AST representation of a program, and `IResourceVisitor`, for traversing trees of resources such as files, folders, and projects. These types, along with the appropriate "hooks" inside the classes representing AST nodes and resources, allow for easy extension of these hierarchies with new methods. However, only these two particular class hierarchies can be extended with new methods, and such extension is only possible because of advance planning by the Eclipse developers. Expanders, on the other hand, allow *any* existing class hierarchy to be easily augmented with new methods, without the need to plan ahead for such extension. Further, they do not suffer from the standard problems of visitors, mentioned in Section 2.1. For example, each expander method can have its own argument and result types.

However, visitors do have one important advantage over expanders. Since each external method is reified as a visitor class, it is easy for external methods to inherit code from one another. A common usage of this ability is to define a visitor that does nothing except recursively visit the children of each node in the given tree. A new visitor class can then subclass from that visitor and inherit the code for traversing the tree "for free," only overriding the particular methods that should perform some useful work. In contrast, method families in an expander are completely unrelated to one another, forcing each to re-implement the basic traversal behavior. We believe this limitation could be solved with a notion of *expander inheritance*. We are currently working on adding support for this feature to the eJava compiler using a compilation strategy similar to the one used to support local overriding expanders.

Second, Eclipse has quite a few *utility classes* that use `static` methods to provide additional functionality to existing classes and interfaces. For example, the `MarkerViewUtil` class provides two methods that manipulate instances of the `IMarker` interface, with the following signatures:

```
public static String getViewId(IMarker m)
public static boolean showMarker(IWorkbenchPage p,
                    IMarker m, boolean showView)
```

This approach to adding new methods to existing types has two main drawbacks. First, callers have to use a verbose and unnatural syntax to invoke such methods, for example `MarkerViewUtil.getViewId(m)`, which clashes with the conceptual intent that the new methods are part of the existing `IMarker` type. Second, if a new method needs to dispatch on the run-time class of the `IMarker` instance in order to determine how to behave, then the programmer must resort to error-prone `instanceof` tests and type casts, as shown in earlier sections.

Expanders naturally solve both of these problems. Figure 30 shows an expander for `IMarker` that adds the two methods described above. These new methods can be invoked using the ordinary calling syntax for methods of `IMarker`. Further, the implementer can provide overriding expanders in order to specify the behavior of each new method for particular classes that implement the `IMarker` interface. The eJava language then does the work of automatically dispatching to the appropriate implementation based on the run-time class of the receiver, as discussed in Section 3.

```
public expander IMX of IMarker {
  public String getViewId() {...}
  public boolean showMarker(IWorkbenchPage p,
                            boolean showView) {...}
}
```

Figure 30: The `IMarker` expander in eJava.

6.2.2 Multiple Implementation Inheritance

Eclipse employs a style in which clients only manipulate an abstraction through an interface, rather than a class. This style cleanly separates interface from implementation, ensuring that clients cannot depend on implementation details and allowing implementers to change the underlying implementations without affecting clients. To aid implementers, Eclipse often provides an abstract class that contains default implementations of a particular interface's methods. Implementers can then easily create a class meeting the desired interface by inheriting from the associated abstract class and overriding methods as appropriate [13]. For example, Eclipse's concept of a *view* provides a visual representation of some content. A view must implement the `IViewPart` interface, which declares or inherits a total of 14 method signatures, and the abstract class `ViewPart` contains default implementations of 12 of these methods, requiring concrete subclasses to implement the other two.

Because Java does not support multiple inheritance, however, a class that subclasses `ViewPart` cannot inherit code from any other classes. This limitation can lead to a significant amount of code duplication. For example, suppose a class must meet another interface in addition to `IViewPart`. It is not possible for it to inherit the default implementations of the other interface; instead, the implementer must explicitly copy these methods in the new class.

Expanders provide a natural solution to this problem. First we create an interface for the two methods that each concrete subclass of `ViewPart` is required to implement:

```
public interface VXRequirements {
  void createPartControl(Composite parent);
  void setFocus();
}
```

Then we create an expander that takes the place of the `ViewPart` class, providing the implementations of the other 12 methods required by `IViewPart`. This expander is shown in Figure 31. Any class that implements the `VXRequirements` interface can now be "expanded" to meet the `IViewPart` interface and "inherit" the default implementations of the associated methods. Other expanders can be used in a similar fashion to allow the class to easily meet other interfaces, without requiring code duplication. Further, the class can still inherit code from a superclass, as usual in Java.

6.2.3 Adapters

Eclipse includes an adapter framework that allows an object to be dynamically extended to meet a new type. A type declares its instances to be adaptable by implementing the `IAdaptable` interface. Clients of that type can implement an *adapter factory* which specifies how to adapt that type's instances to new types, and register this factory with a global registry. An adapter factory from Eclipse, which adapts instances of `IJavaBreakpoint` to `IWorkbenchAdapter` is shown in Figure 32. The adapter factory can then be invoked via the `getAdapter` method of the adaptee object, for example:

```
public expander VX of VXRequirements {
  private IWorkbenchPartSite partSite = null;
  private ListenerList propChangeListeners =
    new ListenerList(1);
  private String title = "", toolTip = "";
  private Image titleImage = null;
  public IViewSite getViewSite() {
    return (IViewSite)getPartSite();
  }
  public void init(IViewSite site) {
    partSite = site;
  }
  public void init(IViewSite site,
                   IMemento memento) {
    init(site);
  }
  public void saveState(IMemento memento) { }
  Object getAdapter(Class adapter) {
    return Platform.getAdapterManager()
                   .getAdapter(this, adapter);
  }
  public
  void addPropertyListener(IPropertyListener l) {
    propChangeListeners.add(l);
  }
  public void
  removePropertyListener(IPropertyListener l) {
    propChangeListeners.remove(l);
  }
  public void dispose() {
    if (!propChangeListeners.empty())
      propChangeListeners = new ListenerList(1);
  }
  public IWorkbenchPartSite getSite() {
    return partSite;
  }
  public String getTitle() {
    return title;
  }
  public Image getTitleImage() {
    if (titleImage == null)
      titleImage =
        PlatformUI.getWorkbench().
                   .getSharedImages()
                   .getImage(ISharedImages.IMG_DEF_VIEW);
    return titleImage;
  }
  public String getTitleToolTip() {
    return toolTip;
  }
}
```

Figure 31: An expander replacing the `ViewPart` abstract class.

```
package org.eclipse.jdt.internal.debug.ui;
public class JavaBreakpointWorkbenchAdapterFactory
  implements IAdapterFactory {
  public Object getAdapter(Object adaptableObject,
                           Class adapterType) {
    if (adapterType != IWorkbenchAdapter.class ||
        !(adaptableObject instanceof IJavaBreakpoint))
      return null;
    return new IWorkbenchAdapter() {
      // ...
      public String getLabel(Object o) {...}
      // ...
    };
  }
  // ...
}
```

Figure 32: An example adapter factory from Eclipse.

```
public expander BPX of IJavaBreakpoint
            implements IWorkbenchAdapter {
  public String getLabel(Object o) {...}
  // ...
}
```

Figure 33: Adapting with an expander.

```
IJavaBreakPoint bp = ...
IWorkbenchAdapter wb = (IWorkbenchAdapter)
  bp.getAdapter(IWorkbenchAdapter.class);
if (wb != null) {...}
```

The adaptee object's `getAdapter` method typically forwards to the global registry, which uses the `getAdapter` method of each adapter factory to search for an appropriate adapter, returning `null` if one cannot be found.

With expanders, we can express both the adapter and its clients much more naturally. Figure 33 provides an expander that adds the appropriate methods to an `IJavaBreakpoint` instance in order for it to meet the `IWorkbenchAdapter` interface. A client of an `IJavaBreakpoint` instance simply `use`s the expander `BPX` in order to implicitly adapt the instance to `IWorkbenchAdapter`, thereby allowing methods like `getLabel` to be called on that instance.

There are several interesting points of comparison between the two approaches. First, it is statically apparent in the declaration of an expander what type is being adapted and to what other type it is being adapted. This makes it easy for clients to understand the intended behavior of an expander. Similarly, it is statically visible through a compilation unit's `use` statements which expanders are being employed by any particular client. In contrast, Eclipse's adapter framework is inherently dynamic. The code of the `getAdapter` method in Figure 32 must be inspected to understand what types are involved in the adaptation. Worse, it is impossible for a client to tell which adapter will be used when it calls `getAdapter` on an object. Of course, such dynamism can also be an advantage, by providing more expressiveness than is supported by expanders. For example, the `getAdapter` method in Figure 32 can easily perform the adaptation in one of several ways, depending on run-time conditions. Eclipse is structured around a dynamic plugin model, so it often requires such expressiveness.

Second, Eclipse's adapters are globally registered, which can easily cause problems. For example, if two plugins each register different adapter factories for the same source and target types, one of these factories will be shadowed, causing one plugin to use an adapter that may not be appropriate for its needs. Since expander usage is statically scoped, there can easily exist multiple expanders for the same source and target types, and different clients can `use` different expanders without conflict.

Third, a type is only adaptable in Eclipse's framework if it is declared to meet the `IAdaptable` interface (and provides a `getAdapter` method). Therefore, the original implementer of a type must plan ahead for adaptation. Unfortunately, this advance planning often does not happen, as indicated by several requests in the Eclipse Bug System [8], shown in Table 1. Fourth, Eclipse's framework allows objects to be adapted to both new classes and new interfaces. In contrast, eJava's expanders allow a class to meet new superinterfaces but not new superclasses.

Finally, some Eclipse adapters are designed to be *stateless*: instead of storing the adaptee object as a field of the new adapter object and delegating to this field as necessary, the adapter object's methods all explicitly accept an adaptee object as a parameter. The example in Figure 32 is in fact a stateless adapter. For example, if `bp` is an instance of `IJavaBreakPoint` and `wb` is the `IWorkbenchAdapter` object returned from the call `bp.getAdapter(IWorkbenchAdapter.class)`, then the label text of the adapted object is accessed by the invocation `wb.getLabel(bp)`.

The stateless adapter design allows one adapter instance to be used across all objects that need to be adapted. In contrast, our compilation strategy for expanders, as discussed in Section 5, always creates one adapter object per adaptee object, thereby consuming more space. However, stateless adapters come at the cost of increased complexity, both for the adapter implementer and the adapter client. On the implementation side, the code for `getLabel` must downcast the given `Object` to an `IJavaBreakPoint` before retrieving its label. On the client side, callers must explicitly manipulate both the adaptee object and its adapter.

6.2.4 External State

Eclipse allows plugins to add new state to existing objects that represent system resources via a notion of *properties*. A property is a name-value pair that can be added to an existing `IResource` instance. Properties are updated and accessed through two methods in the `IResource` interface:

```
void setSessionProperty(QualifiedName key,
                        Object value)
Object getSessionProperty(QualifiedName key)
```

For example, Figure 34 shows how Eclipse's `BuildManager` class manipulates a property `K_BUILD_LIST` on instances of `IProject`.

The property mechanism is useful but has several drawbacks. First, it cannot be used to add new state to objects that are not resources. Second, properties are not statically typed. For instance, nothing prevents two calls to `setSessionProperty` from storing objects of different types to the same property. Since most callers of `getSessionProperty` downcast the resulting object to the expected type of the property, such type disagreement could lead to run-time errors like `ClassCastException`s. Third, each property of an object is globally accessible by all of the object's clients, which can lead to conflicts. For example, two clients could accidentally use the same `QualifiedName` to represent two distinct properties, causing unexpected interactions between these clients at run time.

Expanders allow state to be added to existing objects without

Table 1: A sample of the Eclipse Bug System's "should implement `IAdaptable`" bugs.

Bug ID	*Summary*
12960	All Update Core model objects must implement `IAdaptable`
109138	`IWorkingSet` should extend `IAdaptable`
23032	Make `IEditorReference` adaptable
22452	Would like `ITextViewer` be a supported Adaptable for `AbstractTextEditor` [api]
80671	proxies should be adaptable

```
package org.eclipse.core.internal.events;
public class BuildManager implements ICoreConstants, IManager, ILifecycleListener {
  public ArrayList getBuildersPersistentInfo(IProject project) throws CoreException {
    return (ArrayList)project.getSessionProperty(K_BUILD_LIST);
  }
  public void setBuildersPersistentInfo(IProject project, ArrayList list) {
    // ...
    project.setSessionProperty(K_BUILD_LIST, list);
    // ..
  }
  // ...
}
```

Figure 34: `BuildManager` in Java

suffering from any of these problems. For example, the following expander augments `IProject` instances with a new field of type `ArrayList`:

```
package org.eclipse.core.internal.events;
public expander IPX of IProject {
  public ArrayList buildList = null;
}
```

Clients like `BuildManager` can then `use` this expander and manipulate the new field as it would any other field of the class, as shown in Figure 35. Accesses to the field are now statically typechecked, and different clients can `use` different expanders without any possibility of conflicts.

7. RELATED WORK

Classboxes, a form of module for OO languages, are the closest language construct to expanders of which we are aware. Classboxes were originally developed in the context of Smalltalk [4] but were recently described in an extension to Java [3]. Classboxes allow existing classes to be *refined* to support new fields, methods, and superinterfaces, as in eJava. Classboxes also support the addition of new constructors and inner classes, as well as the overriding of existing methods of the class, all of which eJava does not support.

There are several important differences between expanders and classboxes. First, like expanders, the refinements in a classbox are only available where that classbox has been explicitly imported. While the visibility of a classbox is scoped, any class members added in a classbox are treated semantically as if they were defined in the original class declaration [2]. This semantics can cause the kinds of accidental overriding problems described in Section 3, which elude modular detection. For example, if a class `C` is later refined with a method `m` by some classbox, and separately a subclass `D` introduces such a method as well, the latter method will be considered to override the former even though neither implementer was aware of the other. Aside from potentially causing unexpected behavior dynamically, this accidental overriding can also lose type safety, for example if the two methods differ on the return type. eJava's method lookup semantics and modular type system prevent such accidental overriding and the associated type safety problems.

Second, classboxes are implemented nonmodularly: all classboxes that refine a class must be available when that class is compiled, and these refinements are weaved into the class appropriately. At that point, regular Java typechecking and compilation are performed on the resulting class. This means that any errors in a classbox are not detected modularly. Further, while this implementation strategy avoids the problem of object identity for wrappers, it allows unrelated classboxes to clash with one another, preventing compilation. For example, if two classboxes define a method `m` for class `C`, but with different return types, the methods are merged into a single `m` method that tests the current scope dynamically to determine what code should be executed. If these methods have different return types, the resulting Java code will not typecheck. While this problem could potentially be addressed by a form of name mangling, that strategy would not work if the methods are needed in order to implement particular interfaces. In contrast, because expanders are implemented modularly, two unrelated expanders cannot conflict with one another.

Finally, clients in Classboxes can only import a single version of a class, while clients in eJava can `use` multiple expanders.

The *inter-type declarations* in AspectJ [17] support similar augmentations to a class as do classboxes. However, aspect usage is not scoped: an aspect's inter-type declarations are implicitly available to clients of the original classes. This semantics reflects the fact that aspects are often intended to update the behavior of existing classes in a manner that automatically updates all existing clients, in order to noninvasively evolve existing applications. While this behavior is quite powerful, it can also easily cause accidental relationships and conflicts among logically independent aspects and classes to occur. Finally, AspectJ employs a similar compilation strategy as described above for classboxes and so suffers from similar problems.

The Scala [23] programming language contains a construct called *views*, which are a form of language support for adaptation. Views are essentially programmer-defined functions from one type

```
package org.eclipse.core.internal.events;
use IPX;
public class BuildManager implements ICoreConstants, IManager, ILifecycleListener {
  public ArrayList getBuildersPersistentInfo(IProject project) throws CoreException {
    return project.buildList;
  }
  public void setBuildersPersistentInfo(IProject project, ArrayList list) {
    // ...
    project.buildList = list;
    // ...
  }
  // ...
}
```

Figure 35: `BuildManager` **in eJava**

to another. However, the language infers where a view should be inserted, in order to allow a value of one type to be treated as having a different type. As with expanders, views must be explicitly imported in order to be considered for such inference. In this way, on the client side views are quite similar to expanders. However, on the implementation side views provide no special support for adaptation. For example, in order to augment a type with new methods, the programmer has to implement a view that explicitly creates a wrapper class for values of that type, along with forwarding methods for all of the original methods of that type. In eJava, these details are taken care of by the compiler. Additionally, Scala does not support any form of overriding for views, so programmers must manually implement run-time type dispatch in order to adapt different subclasses in different ways.

MultiJava's open classes [6] allow new methods to be added to existing classes externally. As in eJava, clients in MultiJava explicitly import any new methods they wish to employ, thereby allowing for modular reasoning. MultiJava does not support the introduction of new superinterfaces to existing classes and hence is not able to express object adapters and related idioms. MultiJava also does not support the introduction of new fields to existing classes. The inclusion of superinterfaces and fields in expanders constitutes a large increase in expressiveness over MultiJava and requires significant new work to support modular typechecking and compilation.

Half & Half [1] is an extension to Java that supports the ability to add new superinterfaces to existing classes, as well as a form of multiple dispatch. A wrapper strategy similar to the one we describe is used to compile the new language construct. Unlike eJava, Half & Half does not support the addition of new methods or fields to existing classes. Therefore, a new interface can only be given to an existing class if it already meets all the requirements of that interface.

Several language features, including mixins [5, 12, 25], traits [24], and parameterized modules [11, 18], support the flexible creation of new classes from existing ones. In contrast, expanders flexibly update existing classes in place. Therefore, expanders can be used to adapt existing objects with new capabilities, while the approaches mentioned above cannot. On the other hand, those language constructs provide fine-grained code reuse across independent classes, while expanders do not. They also support some expressiveness that could be useful to incorporate in expanders. For example, Schärli *et al.* [24] describe an expressive sublanguage that allows clients to resolve conflicts that may occur when combining multiple traits. As another example, Jiazzi [18] supports a very expressive form of class parameterization, including recursive linking.

There have also been several languages that include a form of virtual types [27, 26] and related forms of dependently typed classes [9, 10, 19, 21]. These languages allow hierarchies of classes to easily be refined in various ways. Like the languages described above, these approaches allow new hierarchies to be created from existing ones, rather than updating existing hierarchies in place. Therefore, these approaches are not suitable for adapting existing objects to new clients. On the other hand, these approaches allow multiple independent hierarchies to exist in a program, along with a guarantee that instances of the constituent classes will only intermingle with other instances from the same hierarchy.

8. CONCLUSIONS AND FUTURE WORK

We have described the expander, a new construct that provides explicit language support for modularly adapting objects in flexible ways to suit the needs of clients. Expanders can be used to implement a variety of recurring idioms in object-oriented software (such as the visitor and adapter design patterns), in a manner that is more declarative, more extensible, and more amenable to static typechecking. We have instantiated our notion of expanders in eJava, an extension to Java. We have formalized a core subset of the eJava language and proven its modular type system sound. We have also implemented eJava via a novel scheme for modularly translating expanders to Java. Lastly, we demonstrated ways in which expanders can be used to improve real-world object-oriented systems through two case studies.

The exploratory study in Eclipse pointed out a few limitations of eJava that we plan to address in future work. First, while a visitor can easily inherit code from another visitor, this is not true for expanders. It would be useful to consider a notion of *expander inheritance*, which would allow an new expander family to be easily derived from an existing expander family. Second, the exploratory study illustrated a potential use for the ability to augment existing classes with new superclasses, in addition to new superinterfaces. We plan to explore this idea to determine if the resulting gain in expressiveness justifies the added complexity. Finally, we plan to extend the eJava language to handle the features in Java 1.5 by porting the eJava compiler to a recently announced extension to Polyglot for Java 1.5 [22].

9. REFERENCES

[1] G. Baumgartner, M. Jansche, and K. Laufer. Half & Half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Department of

Computer and Information Science, The Ohio State University, revised March 2002.

[2] A. Bergel. Personal communication, Oct. 2005.

[3] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[4] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003. Best Award Paper.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

[6] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

[7] Eclipse home page. `http://www.eclipse.org`.

[8] Eclipse Bug System home page. `https://bugs.eclipse.org/bugs`.

[9] E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, 2001.

[10] E. Ernst. Higher-order hierarchies. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS, Darmstadt, Germany, July 2003. Springer Verlag.

[11] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 94–104. ACM, June 1998.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[13] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[14] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[16] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, LNCS 2072, Budapest, Hungary, June 2001. Springer-Verlag.

[18] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 211–222. ACM Press, 2001.

[19] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 99–115. ACM Press, 2004.

[20] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, Apr. 2003.

[21] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 41–57, New York, NY, USA, 2005. ACM Press.

[22] Polyglot for Java 1.5 home page. `http://www.sable.mcgill.ca/~jlhotak/polyglot-custom`.

[23] The Scala language home page. `http://scala.epfl.ch`.

[24] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[25] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In E. Jul, editor, *ECOOP '98–Object-Oriented Programming*, LNCS 1445, pages 550–570. Springer, 1998.

[26] K. K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 186–204, Lisbon, Portugal, June 1999. Springer-Verlag.

[27] M. Torgersen. Virtual types are statically safe. In *International Workshop on Foundations of Object-Oriented Languages (FOOL) , informal proceedings*, Jan. 1998.

[28] A. Warth and T. Millstein. Featherweight eJava. Technical Report CSD-TR-060013, UCLA Computer Science Department, 2006. `http://www.cs.ucla.edu/~todd/fej.pdf`.

[29] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 Nov. 1994.