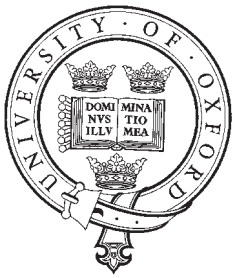
Pure Aspects

Elçin A. Recebli

Wolfson College



University of Oxford
Computing Laboratory

August 2005

Abstract

Aspect-oriented programming (AOP) is a relatively recently proposed programming paradigm that introduces *aspects*. Conceptually, an aspect is a module that observes the flow of a program and performs specified actions upon encountering specified events in the execution. This feature makes aspects a convenient tool for addressing *cross-cutting concerns*,—*i.e.* concerns that systematically affect all or some other components of the whole system.

However, the power comes at the price of increased responsibility. Ability to observe the program flow and alter and/or augment other components also implies the ability to break encapsulation, which is one of the central notions at the heart of the OOP paradigm. Taking into account that AOP is most commonly built on top of OOP, this is a potential source of unforeseen problems. How do we decide when encapsulation violations are to be allowed, and when not? What can be done to automate making decisions?

In this dissertation one approach to solving the problem—the notion of *aspect purity*—is proposed. From a bird's eye view, a *pure aspect* is one that promises not to alter the behaviour of specified set of modules (classes), only possibly adding something new. For example, if certain program models some physical process, then the aspect used to build a visualisation of that process, *e.g.* by creating a graphical window and constantly updating its content as a response to changes of parameters, is a good candidate to be pure: it is not supposed to change the way the process itself is modeled; it is only there to watch certain values and perform corresponding drawings.

Our design provides a seamless extension to the most popular aspect-oriented programming language—*AspectJ*. AspectJ is itself a conservative extension of Java. We then proceed to describe our implementation of these ideas. Our starting point for the implementation is *abc*—an extensible workbench for experimenting with new features of AspectJ.

In large and complex systems, violations of purity by aspects supposed to be pure can be accidental and hard-to-spot. We will see examples of such situations, what harm they can cause, and how automated purity verification could have helped avoiding arisen problems.

Acknowledgments

I would like to express my gratitude to my supervisor, Professor Oege de Moor, for introducing me to the world of AOP and getting me involved with this innovative project, for his assiduous guidance and support.

I am also grateful to all members of abc mailing list for their lively participation in the development of the concept of pure aspects and valuable comments about this matter. Thank you all!

Last, but not least, I am thankful to my wonderful family for their constant countenance and sincere concern that I could almost physically feel during the last year, and that never let me down.

Contents

Chapter 1

Introduction

1.1 Motivation

Aspect-Oriented programming, or AOP for short, is a paradigm that introduces a new kind of program modules, namely *aspects*, specifically designed to address cross-cutting concerns.

An aspect, like any module, *e.g.* a class, is a part of a larger system. However, unlike a class, it is not an entity in itself, but rather a component that cross-cuts all or some other components, altering the existing design in a systematic way. It does so by observing a running program and triggering specified actions—*pieces of advice*—on encountering specified events in the program flow.

These powerful features make aspects capable of solving many problems in a more elegant and effortless way than would be possible with plain OOP. Clear separation of cross-cutting concerns has a positive impact on program maintainability.

However, the ability to alter the behaviour of existing classes conflicts with encapsulation—one of the central notions of OOP. Overall AOP nicely builds on top of the latter, but at the same time the very core of these two paradigms don't seem to get on well with each other, at least at first sight.

A potentially troublesome area of AOP, following from the ability to arbitrarily break encapsulation, is impaired modularity of systems. According to Parnas [1], one of the benefits of decomposing a system into modules should be enabled *modular reasoning*,—*i.e.* the possibility to understand the whole system one module at a time. With aspects' potential to burst into the code of other modules this is often

impossible: when studying some module, one would need to consider all aspects that could possibly interfere and change the module's logic.

So how can we deal with these unpleasant situations?

1.2 Contribution

We need some foundation to resolve possible collisions of OOP and AOP. Clearly, not all breaches of encapsulation by aspects are harmful—otherwise AOP would be useless. Ultimately what we want is a criterion that will let us distinguish between "good" and "bad" violations of encapsulation.

Such criteria can come in two flavors: either a class can explicitly advertise what implementation details external aspects are allowed to see, and at what points they are allowed to intervene; or an aspect can declare what properties of the base system it is, or is not expected to alter.

Both approaches have their own scope and can be applied to address specific issues. The universal solution probably has to provide means to formalise requirements from either side.

The aim of this dissertation is to propose one method of controlling behaviour of aspects from their side, and implement that method as an extension to *abc*—a compiler for *AspectJ*, which is an AOP extension to Java programming language. Special attention is given to considerations leading to introduction of the notion of aspect purity.

In particular, the contributions of this dissertation are as follows:

- a careful analysis of the ways aspects can break encapsulation
- a proposal for a new language feature of AspectJ, namely *purity annotations* that overcome these problems without sacrificing flexibility
- an implementation of purity annotations in abc, using sophisticated analysis to verify purity
- an evaluation of our design and implementation on a number of non-trivial examples

1.3 Outline

Chapter 2 provides an introduction to aspect-oriented programming.

In Chapter 3 a new classification of aspects based on their intended roles in the system is introduced. After that we will see examples of aspects that render the system erroneous, because they unintentionally cross the boundaries of classification they belong to. This serves as a motivating example for proposal behind this dissertation.

In Chapter 4 several ways of limiting applications of aspects are discussed, and a new approach—the concept of *pure aspects*—is proposed and justified. We will see how this proposal can help solving problems exposed in Chapter 3, and what other applications it has.

Chapter 5 gives an account of how these new features have been implemented in abc, an extensible compiler for AspectJ. Basic principles behind the architecture of abc are described and how they serve our aims. The design and implementation of abc extension for support of pure aspects are described in detail.

Chapter 6 shows the results of experimenting with the new extension. Finally, Chapter 7 gives an account of related work, and Chapter 8 draws conclusions and outlines possible directions for future development.

Chapter 2

Aspect-Oriented Programming

2.1 Introduction to AOP

Aspect-oriented programming [2, 3] is a relatively recently proposed paradigm, which introduces a new kind of modules—*aspects*—designed to address cross-cutting concerns.

A *concern*, generally speaking, is a sub-goal, a specific requirement that constitutes a part of the whole *software system*, which in turn consists of a set of concerns. For example, an (oversimplified) electronic shop is a realisation of the following concerns: customer registration, sales records, warehouse management, credit card processing, goods shipping, security. The last one is of special interest for us here, because it is a *system-level concern*: security must be maintained in every part of the system, in every module, so corresponding code bits would be scattered throughout other modules. Thus, it *cross-cuts* the whole system. In [4] R. Laddad shows how such security policy can be localised in a single aspect.

In general, the following classification criterion has been been proposed by Gregor Kiczales *et al.* in [2]: if a concern cannot be cleanly encapsulated in a procedure/-function/class (depending on the type of target programming language), but rather affects a set of procedures/functions/classes in a systematic way, then it should be represented via an aspect.

Aspects are intended to make implementing such concerns easier and more natural. Utilising concepts of AOP results in better modularisation, cleaner code, and hence improved maintainability. Returning to our example of an e-shop, consider

details of how security concern could be implemented using OOP language like Java: security-related code, *e.g.* user name, password, access level verifications, should be inserted at every method that required such checks, leading to tangled code. In contrast, AOP would allow to implement this concern in one place in a separate aspect module, and define rules dictating how the aspect code should be fused into the base code.

To achieve its goal, in addition to *component language*—the one used to program regular concerns—AOP-based approach requires *aspect language*, with which to program aspects. Also required is an *aspect weaver*—a program that fuses aspect code into the base code following instructions provided. In practice *weaving* can be done either at run-time, or at compile-time. The elements of the component language semantics that aspects coordinate with are called *join points*.

2.2 AspectJ: AOP & Java

AspectJ [5, 6] is a simple and practical general-purpose aspect-oriented extension to Java. It is designed to be compatible with Java, the word *compatible* embodying 4 things:

Upward compatibility meaning that all legal Java programs should remain legal AspectJ programs and retain their semantics

Platform compatibility meaning that AspectJ programs must compile into regular *.class* files, runnable on a standard JVM

Tool compatibility it must be possible to extend existing Java development tools to support AspectJ development in a natural way

Programmer compatibility it must be easy for Java programmers to quickly learn AspectJ-specific features, which must feel as natural as possible in Java environment

AspectJ was originally developed by G. Kiczales *et al.* and later became a part of Eclipse open-source project [7], supported by IBM. Mainstream AspectJ compiler *ajc* is built on the basis of Eclipse Java compiler.

2.2.1 Examples

Before proceeding to describing the language elements more in detail, let us have a look at what real benefits programming in AspectJ can offer.

Logging

Logging is a prevalent technique used for several purposes. During development cycle, logging can help to better understand program behaviour and hence simplify discovering bugs. In security sensitive systems logging can be used for security reasons.

Logging is about printing or saving messages about performed operations. Standard Java API includes `Logger` class specifically designed for implementing logging. This class provides convenient ways for various logging tasks. Nevertheless, it still requires appropriate code to be inserted in every method:

Listing 2.1: Logging with Plain Java

```
1  import java.util.logging.*;
2
3  public class AClass {
4      static Logger _logger = Logger.getLogger("trace");
5
6      public void foo() {
7          _logger.logp(Level.INFO,
8                       "AClass", "foo", "Entering");
9          // method code
10     }
11
12     public void bar() {
13         _logger.logp(Level.INFO,
14                      "AClass", "bar", "Entering");
15         // method code
16     }
17 }
```

Besides being cross-cutting, logging is also *invasive*: when a new module is added to the system, all of its methods requiring logging must be augmented with appropriate code.

AspectJ allows to get rid of all this tangling and implement logging concern in a very compact and modular way (copied from [4, pp. 153-154]):

Listing 2.2: TraceAspect performing the same job

```
1  import java.util.logging.*; import org.aspectj.lang.*;
2
3  public aspect TraceAspect {
4      private Logger _logger = Logger.getLogger("trace");
5
6      pointcut traceMethods()
7          : execution(* *.*(..)) && !within(TraceAspect);
8
9      before() : traceMethods() {
10         Signature sig = thisJoinPointStaticPart.getSignature();
11         _logger.logp(Level.INFO, sig.getDeclaringType().getName(),
12                     sig.getName(), "Entering");
13     }
14 }
```

This aspect removes both tangling and invasiveness: it automatically performs required actions at proper points without any need for manual code insertion.

Caching

Consider a simple Java program for computing factorial, taken from [4, pp. 92-93]:

Listing 2.3: TestFactorial.java: factorial computation

```
1  import java.util.*;
2
3  public class TestFactorial {
4      public static void main(String[] args) {
5          System.out.println("Result: " + factorial(5) + "\n");
6          System.out.println("Result: " + factorial(10) + "\n");
7          System.out.println("Result: " + factorial(15) + "\n");
8          System.out.println("Result: " + factorial(15) + "\n");
9      }
10
11     public static long factorial(int n) {
12         if (n == 0) {
13             return 1;
14         } else {
```

```
15              return n * factorial(n-1);
16          }
17      }
18  }
```

The implementation of `factorial()` function is elegant yet inefficient: factorial of one and the same number is computed time and again. We would prefer the function to reuse previously computed values. However, caching for efficiency is a separate concern here: it has its own logic and doesn't change the essence of the factorial computation algorithm, rather it cross-cuts the latter. Using AspectJ's aspect language, we can implement caching in its own module—aspect:

Listing 2.4: OptimizeFactorialAspect.java: aspect for caching results

```
1  import java.util.*;
2
3  public aspect OptimizeFactorialAspect {
4      pointcut factorialOperation(int n) :
5          call(long *.factorial(int)) && args(n);
6
7      pointcut topLevelFactorialOperation(int n) :
8          factorialOperation(n)
9          && !cflowbelow(factorialOperation(int));
10
11     private Map _factorialCache = new HashMap();
12
13     long around(int n) : factorialOperation(n) {
14         Object cachedValue = _factorialCache.get(new Integer(n));
15         if (cachedValue != null) {
16             // Found cached value for n
17             return ((Long)cachedValue).longValue();
18         }
19         return proceed(n);
20     }
21
22     after(int n) returning(long result)
23         : topLevelFactorialOperation(n) {
24         _factorialCache.put(new Integer(n), new Long(result));
25     }
26 }
```

The caching strategy is as follows: save all values returned by a non-recursive call to `factorial()`, together with the passed argument; intercept all (recursive or not) calls to `factorial()` and return saved result if there is one corresponding to given call argument, otherwise proceed to `factorial()`.

Thus, we have completely separated the caching logic from base computations. Now we can change this logic without caring about the `factorial()` method, and *vice versa*, should we decide to do so.

2.2.2 Join Points

In AspectJ, a *join point* is a well-defined identifiable point in the execution of a program. The join point should be identifiable not only in the source, but also in the sequence of bytecode instructions executed at run-time: this is due to the concept that aspects *observe* program flow, so availability of the base program's source code is not a requirement. Examples of join points are method calls, field sets and gets. However, not all kinds of join points are available in AspectJ. For example, exception throwing is an easily identifiable point, nevertheless there is no way in AspectJ specification to refer to them.

All join points *expose* some *context*. For example, instance method calls expose target instance, calling instance and call arguments; exception handling join point exposes caught exception and current instance.

Available join point categories together with context they expose are listed in Table 2.1.

2.2.3 Pointcuts

A *pointcut* is a construct that identifies join points. It can be viewed as a kind of pattern that matches some set of join points. For example, a pointcut can match calls to specified method happening within specified class. Besides this, pointcuts can collect context exposed by join points they match. AspectJ includes pointcut primitives (listed in Table 2.2), from which complex pointcuts can be built using logical operations && (*and*), || (*or*), and ! (*not*). For example, a pointcut

```
call(void *.setTitle(String)) && args(title)
```

Table 2.1: Join Point Categories

Join Point Categories	Exposed Context
Method execution	Target object, arguments, return value
Method call	Current and target objects, arguments, return value
Constructor execution	Constructed object, arguments
Constructor call	Current object, arguments
Class initialisation	none
Field read access	Reading object, target object, field value
Field write access	Writing object, target object, set value
Exception handler execution	Current object, exception object
Object initialisation	Instance, constructor arguments
Object pre-initialisation	Constructor arguments
Advice execution	Aspect instance, advice arguments, return value

matches calls to `setTitle(String)` method of any class and binds the value of the single passed argument of type `String` to `title`. A pointcut

```
set(int Point+.x) && target(point) && args(x) && if(x < 0)
```

matches writes to an integer field `x` of any object of type `Point` or any of its (direct or indirect) subtypes, if the value being set is less than 0.

Obvious difference between the two examples is that in the first case the pointcut matches join points statically, while in the second matching can only be determined at run-time.

2.2.4 Advice

An *advice* is a piece of code that is executed at join points selected by a pointcut. The execution can take place *before*, *after*, or *instead of* the join point. Advice can also manipulate values collected by its pointcut at the join point.

The following piece of advice is a (naïve) realisation of a concern that frame title should always reflect the name of the frame's background color:

Table 2.2: Pointcut Primitives

Pointcut Primitive	Join Point
execution(MethodSignature)	Method execution
call(MethodSignature)	Method call
execution(ConstructorSignature)	Constructor execution
call(ConstructorSignature)	Constructor call
staticinitialization(TypeSignature)	Class initialisation
get(FieldSignature)	Field read access
set(FieldSignature)	Field write access
handler(TypeSignature)	Exception handler execution
initialization(ConstructorSignature)	Object initialisation
preinitialization(ConstructorSignature)	Object pre-initialisation
adviceexecution()	Advice execution
cflow(Pointcut)	All join points in the control flow of the join points matched by specified pointcut
cflowbelow(Pointcut)	Like **cflow**, but excluding the join points matched by specified pointcut
within(ClassnamePattern)	All join points within specified class
withincode(MethodSignature)	All join points within specified method
this(ClassnamePattern)	All join points at which current object's type matches ClassnamePattern
this(AdviceFormalName)	Used to bind current object to advice formal
target(ClassnamePattern)	All join points at which target object's type matches ClassnamePattern
target(AdviceFormalName)	Used to bind target object to advice formal
args(TypenamePattern(s))	All join points with type(s) of call argument(s) matching TypenamePattern(s)
args(AdviceFormalName(s))	Used to bind call argument(s) to advice formal(s)
if(condition)	All join points at which `condition` is satisfied

```
after(Frame frame, Color bgColor): call(Frame+.setBackground(Color))
                                && args(bgColor) && target(frame) {
    if (bgColor == Color.BLUE)
        frame.setTitle("Blue");
    else if (bgColor == Color.RED)
        frame.setTitle("Red");
    else if (bgColor == Color.GREEN)
        frame.setTitle("Green");
    ...
}
```

Of special interest are pieces of advice that get executed instead of join points. In AspectJ these are implemented by *around advice*. It is possible to call the predefined `proceed()` function, which indicates that the piece of code replaced by the around advice must be executed. **proceed()** expects arguments corresponding to every element of context collected by the pointcut, which may or may not be equal to actual collected context. **proceed()** returns, and the advice must also return, a value of the same type as the one returned by the join point.

A concern that a point's coordinates must never be set to values less than 0 can be expressed via an around advice:

```
void around(int coord): set(int Point.*) && args(coord)
                                && if(coord < 0) {
    proceed(-coord);
}
```

So an aspect is basically a collection of pairs of form $(pointcut, advice)$. Besides, it can have an internal state by maintaining fields, exactly like classes in Java.

To sum things up, in AspectJ one can view an aspect as a construct that observes the flow of a program and triggers specified actions upon encountering specified events. This illustrates how cross-cutting concerns are addressed by aspects: if a particular concern is a system-level one, in plain object-oriented or procedural style the code corresponding to its implementation would be scattered throughout the whole program; aspects can accumulate all such bits of code in pieces of advice and

automatically execute them when required.

Listing 2.5 accumulates some of the concepts described in this section:

Listing 2.5: Example of an AspectJ Program

```
1  class XmplClass {
2      private int j = 9;
3
4      public static void main(String[] args) {
5        XmplClass xc = new XmplClass();
6        xc.bar(args);
7        xc.foo(null);
8      }
9
10     public void foo(Object o) {
11       System.out.println("foo()...");
12     }
13
14     public void bar(Object o) {
15       System.out.println("bar()...");
16       j = 10;
17       System.out.println(sum(5));
18     }
19
20     public int sum(int i) {
21         return j + i;
22     }
23 }
24
25 public aspect XmplAspect {
26     pointcut stat(): execution(void *.f*(..));
27     /* void return type, any class name, methods with names
28         starting with 'f' */
29
30     pointcut dyn(Object o): execution(!static * XmplClass.*(..))
31         && args(o,..) && if(o instanceof String[]);
32     /* non-static, any return type, XmplClass class, any method,
33         collect first argument and bind to variable o */
34
35     after(): stat() {
36         System.out.println("static join point matching");
37     }
38
```

```
39      before(Object o): dyn(o) {
40          System.out.println("argument of type String[]");
41      }
42
43      pointcut arnd(int i, XmplClass x): call(int XmplClass.sum(int))
44          && args(i) && target(x);
45      /* calls to XmplClass.sum(int) method, bind passed argument to i
46          and target instance to x */
47
48      int around(int i, XmplClass x): arnd(i, x) {
49          return proceed(i, new XmplClass()) + 1;
50          /* call proceed() with replaced target object and add 1 to
51              the result */
52      }
53  }
```

Pointcut `stat()` matches bodies of all methods whose names start with 'f', regardless of the run-time context, so this is an example of situation when the advice body can be inserted straight away at the join point.

Pointcut `dyn()` shows how run-time context is collected. Moreover, it matches in those cases when a method is passed an argument of type `String[]`, which can only be detected at run-time. Hence this pointcut causes a dynamic run-time test to be inserted, which decides whether or not to execute advice body.

The result of running this program is below:

```
argument of type String[]
bar()...
15
foo()...
static join point matching
```

Chapter 3

Harmful Aspects

3.1 Classification of Aspects

Aspects can be classified in different ways. One classification based on kinds of interactions between pieces of advice and methods is given in [8]. In [9] aspects as a whole are considered and classified. These and similar works present a more or less low-level perspective dealing explicitly with what individual pieces of advice may or may not do.

Here we propose a different approach: our classification is based on intended roles of aspects in software systems from higher-level engineering point of view.

Definition 3.1 Integral*, or* tightly bound aspects *are those that implement integral concerns of systems they belong to. They provide core functionality; when removed they cause the system to work improperly and may even render it uncompilable.*

Attachable*, or* loosely bound aspects *are those that can be attached to otherwise fully functional systems in order to provide additional and/or alter existing functionality.*

Integral aspects can be exemplified by an aspect responsible for output of a graphical editor: the latter would be meaningless without proper GUI. Such aspects are more like classes designed to address cross-cutting concerns.

Logging and caching aspects, visualisers are usually attachable. This kind of aspects are like software plug-ins and extensions in generally accepted sense [10]. General-purpose aspect libraries are likely to consist mainly of attachable aspects.

Listing 3.1 gives a concrete example of an integral aspect (**privileged** keyword in aspect declaration means that it may access private fields and methods):

Listing 3.1: Integral Aspect

```
1  public class AClass {
2      int state = 0;
3
4      public static void main(String[] args) {
5          AClass a = new AClass();
6          a.foo();
7          a.bar();
8      }
9
10     private void foo() {
11         System.out.println("foo()...");
12     }
13
14     public void bar() {
15         if (state == 0)
16             throw new IllegalStateException();
17         else
18             bar1(); // introduced by the aspect
19     }
20
21 }
22
23 public privileged aspect IntegralAspect {
24     public void AClass.bar1() {
25         System.out.println("bar1()...");
26     }
27
28     after(AClass a): initialization(AClass.new(..)) && this(a) {
29         a.state = 1;
30     }
31 }
```

In the above example the program would not compile without the aspect, since `bar()` method calls `bar1()`, which is introduced by the aspect (this is called *static cross-cutting*, as opposed to *dynamic cross-cutting* via pieces of advice). Furthermore, the aspect performs necessary initialisation, without which the program would throw an exception.

Example of an attachable aspect is given in Listing 3.2, taken from R. Laddad's "AspectJ in Action" [4, pp. 37-38]:

Listing 3.2: Attachable Aspect

```
1  public class MessageCommunicator {
2      public static void deliver(String message) {
3          System.out.println(message);
4      }
5
6      public static void deliver(String person, String message) {
7          System.out.print(person + ", " + message);
8      }
9  }
10
11 public aspect MannersAspect {
12     pointcut deliverMessage()
13         : call(* MessageCommunicator.deliver(..));
14
15     before() : deliverMessage() {
16         System.out.print("Hello! ");
17     }
18 }
19
20 public class Test {
21     public static void main(String[] args) {
22         MessageCommunicator.deliver("Wanna learn AspectJ?");
23         MessageCommunicator.deliver("Harry", "having fun?");
24     }
25 }
```

The test program would perfectly run and do its job without `MannersAspect`; the aspect only makes it a bit more polite when woven ("attached").

There are also mixed cases possible: an aspect can be integral part of one set of classes, yet only use another set classes to gather information without actively interfering with them.

Integral aspects are usually written as parts of specific systems, hence they are unlikely to be ever applied to others causing unexpected consequences. From now onwards we will be focusing our attention on attachable aspects.

3.2 When Aspects Do Harm

Attachable aspects, being more general in nature, can in some situations have unexpected and undesirable side-effects.

Assume that we have a 3D modeling application. Modeled objects are a table and a smaller object, say, a glass, on the table. When the table is moved, the glass also moves; the latter can move autonomously.

This concept can be captured in the following piece of code:

Listing 3.3: A Table and a Glass Example

```
1  public class Table {
2      private Glass glass;
3      public int x, y;
4
5      public Table(Glass glass) {
6          this.glass = glass;
7          x = 50;
8          y = 50;
9          glass.x = 50;
10         glass.y = 50;
11     }
12
13     public void move(int dx, int dy) {
14         x += dx;
15         y += dy;
16         glass.move(dx, dy);
17     }
18
19     public Glass getGlass() {
20         return glass;
21     }
22 }
23
24
25 public class Glass {
26     public int x, y;
27
28     public void move(int dx, int dy) {
29         x += dx;
30         y += dy;
31     }
```

```
32 }
33
34
35 public class Model {
36     public static void main(String[] args) {
37         Glass glass = new Glass();
38         Table table = new Table(glass);
39         glass.move(10, 10);
40     }
41 }
```

The table "knows" the glass, because when moved it must also move the latter. The glass, however, can move independently, so it doesn't maintain a reference to the table object.

Suppose now that the task is to stick the glass to the table, *i.e.* make the table move when the glass is moved. A careless or hasty programmer may be quick to write an aspect like this:

Listing 3.4: GlassAspect

```
1  public aspect GlassAspect {
2      Table table;
3
4      after(Table t): initialization(Table.new(Glass))
5              && target(t) {
6          table = t;
7      }
8
9      after(Glass g, int dx, int dy):
10         execution(void Glass.move(int, int))
11             && args(dx, dy) && this(g) {
12         if (table.getGlass() == g)
13             table.move(dx, dy);
14     }
15 }
```

This aspect collects the reference to the table immediately after construction and saves it locally. Then it advises the `move()` method of the glass to also move the table.

Although the code compiles without any warnings, any attempt to run the augmented program will result in `StackOverflowError` being thrown. This is quite natural,

since we now effectively have two methods infinitely calling one another.

The issue in the above example is that the aspect changes internal state of the instance of the class `Glass`: while moved by a given displacement, the glass in effect attempts to move infinitely and carry the table away. We did not want to change the existing behaviour of the glass, we only wanted to change the behaviour of the table to make it follow the glass. The glass should move exactly the way it used to; it is the table that should be amended.

In large projects similar things can happen quite often and can be buried deeper into the source code thus becoming harder to spot. This kind of behaviour is not necessarily harmful: it's a common practice to use pieces of advice in ways that actually provide for significant alteration of behaviour of classes, and situations like the one in Listing 3.4 can be avoided.

Nevertheless, there's currently no way for a developer to state that a particular aspect is not expected to change any of existing functionality of a module, but rather to add something to it non-destructively.

Chapter 4

New Safety Features

4.1 Limiting Aspects: Previous Proposals

We need some foundation to prevent harmful breaches of encapsulation by aspects. This problem can be tackled from two sides.

First, classes may explicitly advertise what parts of them can be advised by what aspects. This approach limits *obliviousness* of aspects, which is a proposition that aspects should not make any suggestions about, or prepare any hooks for, the aspects that are applied to them. Obliviousness [11] has been traditionally viewed as a desirable property of modules in AOP.

Katz [12] has proposed that some kind of specification be attached to every module to which aspects can apply, describing all properties that are essential to that module and must not be changed by any aspect. These can be formally specified in a programming language like *Eiffel* [13], or in *JML* [14]—a behavioural interface specification language for Java. While this seems to be a reasonable limitation of obliviousness, it will require a good amount of work for formalising all the specifications.

Another perspective is proposed by Jonathan Aldrich in the form of *open modules* concept [15]. This approach suggests combining classes and aspects in *modules* (the word "module" here bearing a meaning distinct from common usage), which export functions and pointcuts. Afterwards third-party aspects are only allowed to advise external calls to the exported functions, or on the exported pointcuts, but not on calls that are internal to the module. Thus, obliviousness is limited by a third entity

specifically intended for this reason, so better modularisation is achieved.

Clifton and Leavens [9] have proposed that classes explicitly accept assistance from aspects. According to their definition, *assistance* is any change in the behaviour of a system introduced by an aspect. For better modularisation they suggest that all acceptances be combined in *aspect maps*.

Second, an aspect may explicitly limit its own role in the system. The most naïve and straightforward way to achieve this is to include name patterns matching those classes inside which an advice is or is not supposed to apply, as a part of every pointcut using **within()** and/or **withincode()** constructs. This approach is suitable when we know exactly what classes are to be advised, and are aware of their implementation, but in other cases it is not applicable.

Several authors [16, 17, 18] advocated attaching specifications to aspects indicating what is assumed about join points at which some advice applies, and what that advice guarantees after being executed. However, as also recognised by Katz [12], since many aspects implement non-functional concerns like security, or fault-tolerance, building their formal specification is quite difficult.

In above cited [9] Clifton and Leavens propose that certain aspects explicitly declare their intention to only *view* the systems they get applied to without actively interfering with the logic of the latter. The authors call such aspects *spectators*. A good example of such aspect is a visualiser that simply provides a graphical representation of undergoing calculations.

In [19] Walker and Dantas discuss the notion of *harmless advice*—an aspect-oriented advice that is constrained to prevent it from interfering with the underlying computations. In that paper, they develop a simple object calculus for the formalisation of the proposed concept.

Harmless advice would be, according to classification proposed by Rinard, Sălcianu, and Bugrara in [8], an *augmentation*, at most *observing* one. However, in languages like AspectJ aspects are more than just a collection of pieces of advice, so their ability to maintain an internal state also needs to be given consideration to.

4.2 Pure Aspects: an Idealisation

We would like to be able to distinguish aspects that either don't change the logic of the modules they get applied to, or at most add some new functionality without

altering the old one.

For example, the caching aspect from Listing 2.4 does not change any existing logic. Normally, it has no visible side-effects except decreased execution time of the advised system.

The logging aspect from Listing 2.2 also does not alter any existing behaviour; it just non-destructively adds logging capabilities to the advised classes.

Recapitulating the example from Section 3.2, we wanted to make the table follow the glass,—*i.e.* the intention was to only change the behaviour of one module—the table; we didn't intend to change the existing behaviour of the glass by turning its finite movements into infinite ones.

Let us now summarise all these in the definition:

Definition 4.1 *We shall call an aspect whose only possible visible effect is added new functionality, but not altered old one,* pure.

In the light of Definition 3.1 we can see that pure aspects are a special case of attachable aspects. They can be correlated to plug-ins, rather than extensions [10].

This is an ideal definition, because it leaves important questions open: what is "behaviour" exactly, and, more important, what does it formally mean to add new functionality to a module, or a whole program? Though in most situations we can answer these questions intuitively, we still need a more solid foundation.

There's little use in this definition on its own. We would prefer to be able to define a mechanised procedure for compile-time purity checking: it could automatically verify purity and report possible violations to avert troubles like the one in Section 3.2.

4.3 Towards a Mechanised Approximation

The need for a mechanised purity verification brings us to the question, what program functionality formally is, and what exactly does it mean for an aspect—or another module for that matter—to add new functions without changing any of existing ones. In general this is quite complicated problem which doesn't have commonly accepted solution yet—it would require formal specification of all parts of the program. Furthermore, even in the presence of such specification, there's little hope of a mechanised check.

However, some prior considerations may ease our case.

4.3.1 Termination

A non-terminating advice applying at otherwise terminating join point will prevent subsequent code from executing. Hence, if some part of system's functionality depends on that code it will be impaired, so the aspect declaring such advice is not pure.

Whatever the case, the halting problem is in general undecidable (first proven by Alan Turing in 1936). This means that we will not always be able to mechanically detect impure behaviour resulting from non-terminating advice application. Static analyses for proving termination are the subject of current research, and there are no obvious candidate algorithms for the task in hand.

4.3.2 Exceptions

An uncaught exception thrown during the execution of an advice impairs all functionality dependent on the code not executed because of the exception. So, theoretically, any statement capable of producing a run-time exception or error is potentially impure. Moreover, even if we disallow all such statements (a draconian measure by itself), any aspect will still have at least some memory overhead, which may result in `OutOfMemoryMemoryError`, whereas no such error would be thrown without the aspect.

Thus, our decision is to only disallow explicit throws. One could argue that all *checked* exceptions must be forbidden, but we feel such a restriction is not very useful in practice, as the advice has to declare (in its **throws** clause) all the checked exceptions anyway.

4.3.3 `around()` Advice

around() advice gets executed instead of a join point, so it suppresses functionality dependent on the join point code. To restore it in the original form, the advice must call **proceed()** exactly once and pass it the same values that were collected at the join point (refer to Section 2.2.4). Besides, the value returned by **proceed()** must be passed on unmodified by the advice in order not to affect the functionality depending on that value.

Hereafter we shall call **around()** advice obeying these rules **proceed()**-*pure*, and aspects all of whose pieces of **around()** advice, if any, are **proceed()**-pure **proceed()**-*pure aspects*.

proceed()-purity can be reasonably approximated automatically by analysing the control flow graph.

It is worth noting that **proceed()**-purity is not a necessary condition for overall purity: the aspect may include code that fully replaces suppressed join point. This is the case with the caching aspect from Listing 2.4, which is not **proceed()**-pure. However, its internal logic is too complicated for it to be automatically recognised as having no visible side-effects at all. Although the aspect fits Definition 4.1, it cannot be classified as pure by a mechanised procedure.

4.3.4 Mutated Locations

We continue by discussing in what other ways existing behaviour can be changed by an aspect.

Let us think of programs as collections of modules in general sense (classes, packages *etc.*), and of purity with respect to individual modules. The behaviour of a module depends on values, references to, or copies of which exist within, or can be obtained by, that module. These include intra-program values (such as module's local variables, global variables, method formals and actuals) and extra-program values maintained elsewhere but obtainable by the module (such as current time). An aspect can change any of these. In fact, even dummy advice "changes" current time, since it has some time overhead. On the other hand, a logging aspect, which intuitively is definitely pure (since it merely watches and records) creates a log file, and the module it is woven into can somehow depend on directory listing, which would include that log file. So in order not to be distractively restrictive, we will ignore changes to extra-program values.

Further, if module M_1 is known to use (and hence depend on) global value v defined in module M_2, and aspect A alters v, does this mean that A changes the behaviour of M_1? What if A has nothing to do with M_1 and only applies within M_2? Intuitively, it feels unjustified to classify A as being not pure with respect to M_1. It is definitely impure with respect to M_2, so knowing this one can easily deduce that after augmenting the latter with A the behaviour of all dependent modules can also change.

Same reasoning applies to the case when private value p is created in M_1 and later on a reference to p is passed to M_2: even though aspect A never applies within M_1, its impurity with respect to M_2 may still indirectly affect M_1, since the two modules interact.

In general we propose the following criterion: if aspect A mutates a location, reference to which was collected during advice application within certain module M, or through direct call to any public member of M, then A is impure with respect to M.

The exact number of heap locations dynamically allocated by each module is, generally speaking, unbounded and cannot be predicted at compile-time. To make an approximation, some kind of abstract heap representation is required; for example, each field of object instances of the same type can be represented by a single abstract field, or all instances allocated at a single source code line can be combined in one abstract location.

So, if an aspect mutates an abstract location shared between several modules, it can affect behaviour of all of them, but is classified as impure only with respect to those that provided access to that location.

How mutated abstract locations relate to those, references to which were collected from join points can be approximated. This task requires interprocedural points-to analysis: since in Java all objects are manipulated via implicit pointers, for every mutation in the form `p.field = newValue` happening in the control flow of a certain advice we need to approximate whether `p` can point to a location also pointed to by any pointer collected by the aspect from a join point within a class with respect to which the aspect is declared pure.

4.3.5 Output

Finally, behaviour can be changed at the output stage. If a module prints to `stdout`, `stderr`, or in some other way transmits the results of its work to a client, from the client's point of view an aspect messing with the output will be no different from an aspect messing with the module itself.

In practice it is rarely possible to tell at compile-time whether the aspect will use the same output channel as the module. Besides, even if it does so, it can clearly mark its own output in some way, so that the client is not confused. For example, consider a module whose output is Java source code, and an aspect that

adds comments to code. This is pure behaviour according to Definition 4.1, but a general-purpose automatic purity verifier can hardly distinguish (in terms of purity) the aspect from the one that adds real code instead of comments.

The decision was to disallow I/O operations.

4.3.6 Exclusions

It is possible that certain formal purity violations are in fact not essential. For example, a translator aspect used to localise GUI should be pure with respect to classes it localises, except that it should be allowed to mutate locations corresponding to strings being translated (we know in advance that this will not affect the functionality); or the programmer might want to allow I/O operations knowing that the particular aspect uses I/O "purely".

As a solution, we propose optionally declaring exceptions to purity by specifying the set of join points that should not be checked by a mechanised purity verifier.

4.3.7 Our Proposal

Now let us generalise all these considerations. Recall from Section 2.2.3 that pointcuts can collect context at matching join points. We'll call all context collected by an aspect within certain module M *values collected within* M, and values assigned to by the same aspect *changed values*. For this particular case of determining advice purity we propose the following test:

If a **proceed()***-pure aspect does not explicitly throw exceptions and does not mutate any abstract locations references to which were collected within* M*, then it is pure with respect to* M*.*

Purity of aspects can be compared to humans' interpersonal relations. If Alice entrusts her mailbox password to Bob and he deletes or replaces some messages, then he is dishonest to Alice (not "pure"). If Alice entrusts her mailbox password to Jane, Jane transfers the trust to Bob *bona fide*, who subsequently corrupts mailbox content, then he is dishonest to Jane, but not to Alice. However, Alice should be aware that Bob can fool Jane, and be cautious when transmitting anything to her.

The notion of purity has an important implication. Write S for the original system, A for the aspect, $A(S)$ for the augmented system, and A_S to denote purity of A with respect to all classes of S. In this case, if we limit our observations to events

in a run of $A_S(S)$ happening within S, then at moments immediately preceding and succeeding every such event E the states of instances of all classes constituting S will be exactly the same as if we were running S, disregarding time delays happening as inevitable consequences of executing pieces of advice. Such property can simplify reasoning about the behaviour of augmented systems. Also, if an aspect is pure only with respect to a set of classes, one can be sure that augmented classes can be used in place of originals providing all the functionality of the latter, minding possible impurity of the same aspect with respect to communicating classes.

Purity of aspects can help us dealing with situations like the one in Section 3.2: if we define aspect `GlassAspect` as pure with respect to class `Glass`, we can be sure that described situation will not go unnoticed in compile-time, provided that the compiler used is capable of performing relevant checks and reporting violations.

Also, as noted in Section 1.1, aspects with their uncontrolled ability to change the code of other modules worsen the situation with modular reasoning about the overall system. With pure aspects this is somewhat remedied: when studying particular module, we can disregard aspects that are declared pure to it. Especially helpful in this sense is total purity (*i.e.* purity with respect to all modules.)

4.4 Purity Syntax

AspectJ standard currently doesn't provide any syntax for declaring purity of aspects. To some extent this can be simulated by writing yet another *controller aspect*, which would check that aspects supposed to be pure behave themselves. Nevertheless, such workaround would not do in all cases. For example, it would not be possible to check that **`proceed()`** is called properly within **around** advice.

We propose a small extension to standard AspectJ syntax allowing declaration of purity. For the purpose of declaring an aspect pure with respect to certain classes, we suggest that

```
pure on ClassnamePatternExpr
```

be added before main aspect declaration, including all modifiers, begins. Here *ClassnamePatternExpr* is a standard AspectJ construct that represents all classes and interfaces whose names match specified pattern.

For example, a logger aspect could be declared as in the following listing:

Listing 4.1: Pure Logger Aspect Declaration

```
1 pure on *
2 public aspect Logger {
3     ...
4 }
```

So, `Logger` is declared pure with respect to all classes it may be applied to. Thus, the developer would know that no exiting functionality suffers from this aspect, so it's possible to reason about system components without worrying about `Logger` altering any existing logic.

Another example is a visualisation aspect that views instances of certain class (say, `Subject`) and reflects changes in a graphical window, which is an instance of another class. This aspect is likely to be pure with respect to the former, but not so with respect to the latter, hence it would be declared

```
pure on Subject+
```

indicating purity with respect to `Subject` and all its direct or indirect subtypes.

When needed, the join points to be excluded from purity verification can be specified using **except** keyword:

```
pure on ClassnamePatternExpr
except Pointcut
```

where *Pointcut* represents all join points, violations at which should be tolerated.

Listing 4.2: Exception to Purity

```
1 pure on *
2 except call(void *.setTitle(String))
3 public aspect Translator {
4     ...
5 }
```

Here `Translator` is declared pure on all classes, except that it is allowed to alter collected context via calls to methods called `setTitle()`. So the following would not

Figure 4.1: Purity Declaration Syntax in BNF

⟨*pure-aspect-declaration*⟩ ::= `pure on` ⟨*classname-pattern-expression*⟩
⟨*aspect-declaration*⟩
| `pure on` ⟨*classname-pattern-expression*⟩
`except` ⟨*pointcut-expr*⟩
⟨*aspect-declaration*⟩

count as a violation:

Listing 4.3: Exception at Work

```
1 after(GUIWindow w): initialization(GUIWindow.new(..)) && target(w) {
2     w.setTitle("Translated by the aspect");
3 }
```

Revisiting the example from Listing 3.4, we could rewrite the declaration of aspect `GlassAspect` as follows:

Listing 4.4: Harmful Aspect A

```
1 pure on Glass
2 public aspect GlassAspect {
3     ...
4 }
```

This declaration would allow to avoid the problem described in Section 3.2: proper compiler would report the violation at compile-time, so the situation could be remedied on time.

Chapter 5

Implementation

5.1 abc: an Alternative AspectJ Compiler

In this section we give an account of implementing a compiler extension for the support of purity in AspectJ. Our implementation is based on the *AspectBench Compiler* [20]—an extensible workbench for AspectJ.

The *AspectBench Compiler* (*abc* for short) is an alternative compiler for AspectJ developed by a joint team from Oxford, McGill, and BRICS universities. Besides providing full support for AspectJ 1.2, abc has been specifically designed to be easily extensible. Main design goals behind abc's architecture are [20]:

Simplicity developing new extensions should be a relatively easy task

Modularity both the compiler and the extensions should be modular; developing a new extension should not require changes to the base code

Proportionality the amount of work required to write an extension should be proportional to the amplitude of the extension

Analysis capability the compiler workbench should include comprehensive program analysis framework

All these make abc an very good choice for implementing our purity verifier. Of special interest is the analysis capability: it is the main thing required to implement purity checking.

abc itself is built on two established frameworks. The front-end is based on *Polyglot*—an extensible compiler framework, and the back-end is built on *Soot*—Java bytecode analysis and transformation framework. Polyglot offers convenient ways for customisation of language grammar and semantic analysis; Soot provides appropriate Java intermediate representation, highly suited for aspect weaving and program analysis tasks.

Support for pure aspects is implemented in *purity* abc extension. The implementation required extending both the front-end (for accepting introduced syntax elements) and the back-end (for purity analysis and violation reporting).

5.1.1 Front-End with Polyglot

Polyglot [21, 22] is an extensible compiler front-end toolkit for Java. It can also be viewed as a source-to-source compiler, which parses Java source files, builds the abstract syntax tree (AST), performs core semantic checks required by the language and outputs Java source code. This base functionality is provided by *jlc* ("Java language compiler") and is implemented as an ordered list of *passes* that are performed upon AST.

The base compiler can be extended in a number of ways. First, Polyglot framework includes *PPG* [23]—an extension to *CUP* parser generator [24] that allows grammars to be selectively extended to create parsers for extension languages by providing operations on a CUP grammar, such as adding, dropping, and renaming of productions. Second, Polyglot allows adding new AST nodes, which extend existing nodes and define specific entry points for relevant compiler passes. There are also other powerful means by which extensibility is achieved, such as adding new passes, but they are not essential to this dissertation.

Polyglot achieves high level of modularisation by heavily using interfaces and factories. This extensive use makes it easy to extend or replace various parts of Polyglot.

abc's front-end groups its passes into 12 sets. The final result of running all the passes is a plain Java AST obtained from AspectJ one, and a separate aspect specific information structure called *AspectInfo*. All Java code from aspects is placed in placeholder elements of Java AST; for instance, aspects are converted to classes, and pieces of advice to methods. AspectInfo accumulates the information concerning weaving.

Table 5.1: Jimple Statements

Statement	Description
Core Statements	
NopStmt	no operation
IdentityStmt	used to assign parameter values and `this` reference to locals
AssignStmt	assignment
Intraprocedural Control Flow	
IfStmt	conditional jump
GotoStmt	unconditional jump
TableSwitchStmt	table-based jump
LookupSwitchStmt	list-based jump
Interprocedural Control Flow	
InvokeStmt	method invocation
ReturnStmt	return statement for non-void return types
ReturnVoidStmt	return statement for void return types
ThrowStmt	throws an exception
RetStmt	not used;
EnterMonitorStmt	mutual exclusion
ExitMonitorStmt	

Since AspectJ standard prescribes that aspects should also be weavable into *.class* files, for which the AST is not accessible, weaving is committed to the back-end.

5.1.2 Back-End with Soot

Soot [25, 26] is an extensible Java bytecode analysis and transformation toolkit based around a number of intermediate representations (IR). For the purpose of this dissertation, we are only interested in *Jimple*—a typed, three-address, stackless high-level IR. In fact it's quite close to Java source code and even can be thought of as a kind of canonical form of the latter. Jimple is relatively simple, is it has only 14 kinds of statements instead of over 200 kinds of instructions in Java bytecode. Available statements are listed in Table 5.1

Textual representation of Jimple for "Hello World" is given in the following

listing:

Listing 5.1: Jimple "Hello World"

```
1  public class Hello extends java.lang.Object {
2
3      public void <init>()
4      {
5          Hello r0;
6
7          r0 := @this: Hello;
8          specialinvoke r0.<java.lang.Object: void <init>()>();
9          return;
10     }
11
12     public static void main(java.lang.String[])
13     {
14         java.lang.String[] r0;
15         java.io.PrintStream $r1;
16
17         r0 := @parameter0: java.lang.String[];
18         $r1 = <java.lang.System: java.io.PrintStream out>;
19         virtualinvoke $r1.<java.io.PrintStream:
20             void println(java.lang.String)>("Hello world!");
21         return;
22     }
23 }
```

Similar to how Polyglot does its job in passes, Soot's execution consists of a number of *phases* that themselves consist of *sub-phases*. Sub-phases are either intraprocedural (*i.e.* act on individual methods) or interprocedural (act on the whole set of available classes). Intraprocedural sub-phases extend `BodyTransformer` abstract class, while interprocedural implement `SceneTransformer`

Within Soot, every phase is implemented by a *pack*, so that every pack is a collection of sub-phases represented by one transformer each. A pack may contain either only `BodyTransformer`s, or only `SceneTransformer`s. The behaviour of packs, as well as individual sub-phases, can be controlled through *phase options* that corresponding phases are programmed to accept.

Soot is extensible via adding new phases that perform required transformation or program analysis.

All IRs, including Jimple, can be produced by Soot either from Java bytecode, or source code. For the latter Soot makes use of Polyglot. After creation an IR can be manipulated programmatically using appropriate API and output back into Java bytecode or plain text. Bytecode generation process includes several optimisation procedures.

Final passes of abc's front-end instructs the Soot-based back-end to "jimplify" the produced AST. *.class* files passed as compiler parameters that have no AST representation are also handled by Soot.

Weaving is performed on the Jimple level. The compiler iterates through all weavable classes and in each of them every pointcut associated with a piece of advice is matched against join points where matching can potentially occur. Each such $(pointcut, joinpoint)$ pair generates a *residue*, which is one of *AlwaysMatch*, indicating that *pointcut* matches statically and the corresponding advice should be woven unconditionally, *NeverMatch*, indicating that *pointcut* doesn't match and the advice should not be woven, and dynamic residue, indicating that whether *pointcut* matches or not can only be decided at run-time, so woven advice is preceded by a check to make the decision about executing it at run-time.

Utilisation of such high-level representation as Jimple, purposefully designed for the task of optimising Java programs, is exactly the point that gives abc significant advantage over ajc, which manipulates the bytecode directly. However, this comes at a price of noticeably slower compilation.

5.2 Extending the Front-End

The front-end of abc is built on top of Polyglot, so it completely inherits the extension mechanism of the latter. Hence extending the abc frontend is like extending Polyglot's jlc.

We need to extend existing AspectJ syntax, and get the compiler to properly parse source files with purity annotations and save purity-related information in the generated AST.

5.2.1 AST Generation

Now that new syntax has been defined, we can start writing the extension itself.

The entry point of every abc extension is an `AbcExtension` class that extends `abc.main.AbcExtension`. To keep source code clean, we'll create it in a separate `abc.purity` package.

First step is extending the lexer. abc's lexer is based on *states*, having four states, which reflects that it's intended to deal with four sub-languages. The states are *Java*, *AspectJ*, *Pointcut*, and *PointcutIfExpr*.

Implementing purity extension requires adding three new keywords in various states. Also **aspect** keyword needs to be added to Pointcut state, because purity declaration comes before the word **aspect**, and if it contains an exclusion pointcut, then **aspect** will be encountered in Pointcut state.

This is captured by `initLexerKeywords` of `AbcExtension`:

Listing 5.2: Adding New Keywords

```
1  public void initLexerKeywords(AbcLexer lexer) {
2      super.initLexerKeywords(lexer);
3      lexer.addJavaKeyword("pure",
4          new LexerAction_c(new Integer
5              (abc.purity.parse.sym.PURE)));
6
7      lexer.addJavaKeyword("on",
8          new LexerAction_c(new Integer
9              (abc.purity.parse.sym.ON),
10                 new Integer(lexer.pointcut_state())));
11
12     lexer.addPointcutKeyword("except",
13         new LexerAction_c(new Integer
14             (abc.purity.parse.sym.EXCEPT)));
15
16     lexer.addPointcutKeyword("aspect",
17         new LexerAction_c(new Integer
18             (abc.purity.parse.sym.ASPECT),
19                 new Integer(lexer.aspectj_state())));
20 }
```

Next step involves extending the parser. The grammar for our extension just adds purity declaration, so we need to extend the base AspectJ grammar with a rule for producing pure aspect declarations. This is handled by PPG. We augment the rule for aspect declaration with a new element for production of pure aspects according to purity annotation from Figure 4.1.

Finally, we need to add new kind of AST node—one corresponding to pure aspect declaration. As mentioned earlier, abc inherits the entire front-end extension mechanism from Polyglot, so writing a clean abc extension also requires adherence to rigorous use of factories and interfaces, as it is the case with Polyglot. Therefore, the first step is to define an interface for the new AST node:

Listing 5.3: PureAspectDecl interface

```
1 public interface PureAspectDecl extends AspectDecl {
2 }
```

This is an empty interface, because a pure aspect doesn't need to provide any new functionality in addition to that provided by regular aspects. What this new AST node has to do is to keep information about purity features—class name pattern matching the classes with respect to which concrete aspect being declared should be pure, and pointcut matching exclusion join points,—*i.e.* those at which purity violations should be excused. The following concrete implementation of `PureAspectDecl` captures these:

Listing 5.4: PureAspectDecl class

```
1  public class PureAspectDecl_c extends AspectDecl_c
2      implements PureAspectDecl
3  {
4      ClassnamePatternExpr pureOn;
5      Pointcut except;
6
7      public PureAspectDecl_c(Position pos, boolean is_privileged,
8                              Flags flags, String name,
9                              TypeNode superClass, List interfaces,
10                             PerClause per, AspectBody body,
11                             ClassnamePatternExpr pureOn,
12                             Pointcut except)
13     {
14         super(pos, is_privileged, flags, name, superClass,
15                 interfaces, per, body);
16         this.pureOn = pureOn;
17         this.except = except;
18     }
19     ...
20 }
```

Recall that Polyglot's work is organised in passes. The passes are implemented using Visitor design pattern, hence each node needs an entry point for welcoming visitors. The new node has two additional child nodes, so the method used by visitors must provide access to them for node visitors, otherwise they will not be checked for semantic correctness:

Listing 5.5: PureAspectDecl class

```
1
2     public Node visitChildren(NodeVisitor v) {
3         ClassnamePatternExpr pureOn =
4                 (ClassnamePatternExpr) visitChild(this.pureOn, v);
5         Pointcut except = (Pointcut) visitChild(this.except, v);
6
7         ...
8     }
9     ...
10 }
```

In order to be able to actually use this new node type, we need to subclass abc's default node factory and add a method for creating instances of `PurityAspectDecl`. This is captured in the following piece of code:

Listing 5.6: Extending the node factory

```
1  public interface PurityNodeFactory extends AJNodeFactory {
2      public PureAspectDecl PureAspectDecl(Position pos,
3                                           boolean is_privileged,
4                                           Flags flags, String name,
5                                           TypeNode superClass,
6                                           List interfaces,
7                                           PerClause per,
8                                           AspectBody body,
9                                           ClassnamePatternExpr pureOn,
10                                          Pointcut except);
11 }
12
13 public class PurityNodeFactory_c extends AJNodeFactory_c implements
14         PurityNodeFactory {
15
16     public PureAspectDecl PureAspectDecl(Position pos,
17                                          boolean is_privileged,
```

```
18                                              Flags flags, String name,
19                                              TypeNode superClass,
20                                              List interfaces,
21                                              PerClause per,
22                                              AspectBody body,
23                                              ClassnamePatternExpr pureOn,
24                                              Pointcut except)
25      {
26          return new PureAspectDecl_c(pos, is_privileged, flags, name,
27                      superClass, interfaces, per, body, pureOn, except);
28      }
29  }
```

The node factory is used by the parser (automatically generated by PPG and CUP from supplied grammar file) to build object-oriented AST representation. This way we ensure that all purity-related information finds its place in the tree.

These steps complete the front-end update. When run, our extension accepts declaration of pure aspects and performs required semantic checks. However, they are still treated as regular ones: no special action is taken and no purity checks are performed. In order to implement the latter we need to extend the back-end.

5.3 Extending the Back-End

As described earlier, abc's back-end is based on the Soot framework and uses Jimple intermediate representation.

We need to extend Soot to perform purity verification on aspects declared pure. Each such aspect must be checked for **proceed()**-purity, explicitly thrown exceptions, performed I/O operations, and collected context alteration. The first three are intraprocedural analysis of advice, while the fourth requires interprocedural points-to analysis.

5.3.1 Points-To Analysis

Points-to analysis is a static program analysis intended to estimate the sets of locations to which pointers could point during program execution. For the purpose of the analysis, memory is divided into concrete locations, and for each variable defined in the program the set of such locations to which it may point is computed.

The term *points-to* was coined by Emami, Ghiya and Hendren in [27]. There were several implementations of the analysis. Emami, Ghiya and Hendren proposed context-sensitive, flow-sensitive approach [27], while Andersen [28] implemented both context- and flow-insensitive version. Another principal question is that of dividing the memory: since number of allocated heap locations is potentially unbounded and generally cannot be computed statically, we need some estimation. One approach is to view all objects of the same type sharing one common "location". Another, more precise one, is to identify locations by their dynamic allocation site.

Soot contains *Spark* [29, 30]—a component intended for points-to analysis. The current version performs a context-insensitive, flow-insensitive analysis. The result is available as points-to sets for every pointer in the program. In fact, the points-to information does not depend specifically on Spark. Soot contains default absolutely naïve implementation of analysis engine which yields the result that every pointer may point to every location; Spark suppresses that implementation and provides its own results.

5.3.2 Purity Analysis

As described in Section 5.1.2, Soot's work is organised in phases, represented by sub-phase pack. Phases are either method-level, acting on individual methods, or global-level, acting on entire program. Soot contains two packs relevant to our analysis: *cg* and *wjtp*.

cg stands for *Call Graph* and its job is to construct a call graph required for whole program analysis. When it finishes its run, a call graph becomes available for use by subsequent phases. Different sub-phases of cg provide different ways to construct the call graph, so they are mutually exclusive and only one of them may be enabled at a time. Spark is a sub-phase of cg; besides constructing the call graph, it also generates points-to information for all the pointers in the program

wjtp stands for *Whole Jimple Transformation Pack*; it is empty by default and custom phases for whole program analysis should be added to it.

We need co add a new global-level phase for the purpose of analysing locations mutated by an aspect. Although remaining parts of purity verification are intraprocedural, we will not define additional method-level phases, as this work can be handled by the same global phase.

All whole program phases in Soot, including cg and wjtp are disabled by default.

purity extension requires new analysis sub-phase to be added to wjtp, and this sub-phase in turn needs valid points-to information provided by Spark. Actually, any other phase capable of performing points-to analysis could be used instead; currently alternative analyser called *Paddle* is being developed by Soot group, but it is still unfinished so the decision was to use Spark.

To enable all required phases, `addJimplePacks()` method needed to be overridden in `AbcExtension` class of purity extension:

Listing 5.7: `addJimplePacks()` method

```
1  public void addJimplePacks() {
2      super.addJimplePacks();
3      PackManager.v().getPack("wjtp").add(
4          new Transform("wjtp.purity", PurityVerifier.v()));
5      PhaseOptions.v().setPhaseOption("cg", "enabled:true");
6      PhaseOptions.v().setPhaseOption("cg.spark", "enabled:true");
7      PhaseOptions.v().setPhaseOption("wjtp", "enabled:true");
8      PhaseOptions.v().setPhaseOption("wjtp.purity", "enabled:true");
9      soot.options.Options.v().set_whole_program(true);
10 }
```

According to Soot's (and abc's) coding conventions, `PurityVerifier.v()` returns singleton instance of `PurityVerifier` class, which is used to perform actual purity analysis based on points-to information made available by Spark (or any other analyser).

As required by Soot, `PurityVerifier` extends `SceneTransformer`—base abstract class for all whole program analysis and transformation phases. Entry point to every `SceneTransformer` is its **void** `internalTransform(String phaseName, Map options)` method.

This method has access to all information available within Soot, including points-to information, since cg pack runs before wjtp. All AspectJ specific information accumulated in AspectInfo structure is also accessible.

In abc, after weaving every aspect becomes represented by a regular class, pieces of advice become regular methods, and advice applications become calls to those methods. Thus, the analysis of aspects comes to analysis of regular Java classes and methods.

The overall sketch of the algorithm is as follows. We iterate through all pieces of advice of an aspect declared pure; for **around()** advice we perform **proceed()**-purity verification; all pieces of advice are checked for explicit exception throws; abstract

Figure 5.1: Purity Checking in Pseudo-Code

```
procedure check_purity()
   forall advice in pure_aspects do
      if is_around(advice) then
         check_proceed_purity(advice)
       end if

      check_exception_throws(advice)

      modified_loc.add_all(modified_by(advice))

      forall class matching advice.pure_on
         forall advice_application in applications_within(advice, class)
            collected_loc.add_all(collected_by(advice_application))
         end forall
      end forall

      check_intersection(modified_loc, collected_loc)

      report_warnings()
   end forall
end procedure
```

locations modified by any advice are collected in *modified values set*. Next, we iterate through all classes with respect to which the aspect is declared pure in a search for advice applications, and gather all context collected by advice applications in *collected values set*. Finally, we check to see if any of modified values has a points-to set intersecting with that of any of collected values.

We felt that it would be annoying if the compiler rejected aspects that are judged to be impure, so we have decided to issue only a compiler warning (so compilation can still be completed when a violation is found) rather than a compiler error. All warnings come with indication of code fragment where the violation has taken place.

At conceptual level, the algorithm is presented in Figure 5.1.

Chapter 6

Experiments

Readers may be wondering whether the approximate purity test that we have proposed is too crude. Does it flag warnings too often, classifying aspects as impure while there are no purity violations in reality? In this chapter, we shall demonstrate through a number of examples that in practice, our purity test is quite accurate.

6.1 A Table and a Glass Revisited

We start with testing how the new compiler extension could have solved the problem introduced in Section 3.2. We annotate the `GlassAspect` aspect to declare it pure with respect to the `Glass` class:

Listing 6.1: GlassAspect with Purity Annotation

```
1 pure on Glass
2 public aspect GlassAspect {
3     ...
4 }
```

This way we state that `GlassAspect` must not be destructive towards the `Glass` class. Note that in fact this is not so: moving the glass triggers the aspect that tries to move the table on which the glass resides, and the table, in turn, attempts to move the glass and so on. So the purity of `Glass` is violated indirectly through a chain of calls.

We now run abc with purity extension enabled:

```
java -Xmx512M abc.main.Main -ext abc.purity Glass.java Table.java
    Model.java GlassAspect.java
```

The compiler issues a warning:

```
GlassAspect.java:14: Warning -- Aspect purity violated
            table.move(dx, dy);
            ^----------------^
```

The source code line, which started started the chain of calls resulting in impurity is shown together with the warning.

6.2 **proceed()**-purity

Now we shall test an aspect that violates proceed purity. Let us change `GlassAspect` a bit:

Listing 6.2: **proceed()**-purity

```
1  pure on Glass
2  public aspect GlassAspect {
3      ...
4      void around(Glass glass, int dx, int dy):
5              call(void Glass.move(int, int))
6              && args(dx, dy) && target(glass) {
7          if (glass == table.getGlass()) {
8              table.move(dx, dy);
9              proceed(glass, dx, dy);
10         }
11     }
12     ...
13 }
```

This `around()` advice does not always call **proceed()**, and this situation is reported by the compiler:

```
GlassAspect.java:11-17: Warning -- Possibly proceed()-purity violated
    in around() advice
    void around(Glass glass, int dx, int dy):
    ^---------------------------------------
```

```
    ...
    }
    ^
```

6.3 Exclusions

To check how exclusions from purity work, we'll instruct the compiler not to count violations happening as a result of a call to any `move()` method:

Listing 6.3: GlassAspect with Purity Annotation

```
1 pure on Glass
2 except call(* *.move(..))
3 public aspect GlassAspect {
4     ...
5 }
```

When run, the compiler accepts the aspect as pure and no warnings are reported.

6.4 `TraceAspect`

The logging aspect discussed in Chapter 2.2 is pure. We added `pure on *` to the aspect declaration and tested our extension by running it on the whole set of classes from Section 5.1.3 of [4]. Indeed, no violations were reported.

(Possible indirect purity violation can result from the logger performing I/O operations later on. This is discussed in Future Work section.)

6.5 Compiler Run Times

Turning whole program mode (WPM) in Soot slows down the execution significantly. The table below illustrates this (all estimate times in milliseconds, obtained on 1.60GHz laptop with 512 MB of RAM running Windows XP SP2, JVM 1.4.2):

Table 6.1: Compiler Run Times

Example	Plain abc	abc in WPM, no purity	abc with purity
GlassAspect	4076	64088	65600
TraceAspect	4446	65022	65198

The time overhead of running purity check itself is almost negligible compared to the time overhead of putting Soot in whole program mode to obtain points-to information.

Chapter 7

Related Work

7.1 Assistants and Spectators

Curtis Clifton and Gary Leavens [9] propose dividing aspects into two categories, *assistants* and *spectators*, for the purpose of enabling modular reasoning in AOP. *Assistants* are the aspects that change the behaviour of modules, while *spectators* are the ones that don't. To facilitate modular reasoning, they suggest that assistance should be explicitly accepted by a module. Once assistance of an aspect is accepted by a module, the aspect is allowed to advice the module in question and any other module used by it. There's no such requirement for spectators, since they are only supposed to *view* modules rather than actively *advise* them.

Spectators are quite close to our concept of pure aspects. However, "spectatorness" is always absolute,—*i.e.* mixed cases when an aspect passively views module A and actively *advises* module B are not given consideration.

7.2 Classification Systems for Advice

Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara [8] present classification systems for kinds of interactions between pieces of advice and advised methods. According to them, an advice is classified as *augmentation*, if the entire body of a method always executes after advice application; as *narrowing*, if the method body either executes in its entirety, or doesn't execute at all; as *replacement*, if the advice replaces the method body; or as *combination*, if the advice and the method body combine in some other way.

They define *scopes* as sets of locations accessed by a method or an advice. Scopes of a method and an advice are said to be *orthogonal*, if the advice and method access disjoint locations; *independent*, if neither the advice nor the method may write a field that the other may read or write; the advice scope is said to *observe* the method scope, if the advice may read one or more fields that the method may write; the advice scope is said to *actuate* the method scope, if the advice may write one or more fields that the method may read; scopes are said to *interfere*, otherwise.

Concept of aspect purity defined in this dissertation fits these classification systems to some extent. In particular, a pure aspect would consist solely of pieces of augmentation advice, whose scopes at most observe scopes of advised methods. The inverse, however, is not true: pointer to certain memory location can be obtained by an advice application in one method, and the location be mutated by another advice application in completely different module; thus, the pieces of advice could both be augmentation observational ones, but the aspect as a whole would not be pure.

7.3 Harmless Advice

Walker and Dantas [19] introduce the notion of *harmless advice*—an aspect-oriented advice constrained to prevent it from interfering with the underlying computations. They say that

> computation A does not *interfere with* computation B if A does not influence the final value produced by B.

A harmless advice is supposed to obey a weak non-interference property,—*i.e.* it may change the termination behaviour of advised computations and use I/O, but it doesn't otherwise interfere with the mainline code. They proceed to define a simple calculus for the formalisation of the notion.

Chapter 8

Conclusions and Future Work

8.1 Summary

Modular reasoning about aspect-oriented programs is restrained by aspects' potential to break encapsulation. This potential also enables aspects to uncontrollably alter existing behaviour of other components of the system. In this dissertation we have discussed possible harm resulting from application of unfriendly aspects and reviewed various proposals of a number of researchers that could help enabling modular reasoning in AOP and preventing harm from aspects. Following that we presented our own perspective on the matter and proposed purity annotations—one possible way of preventing unforseen misbehaviour of aspects.

Ideally, a pure aspect is the one that acts non-destructively with respect to other modules and does not impair any of existing functionality. We have also shown how our idealised vision can be made approximable by a mechanised purity verification routine and of what benefit such a routine can be to a programmer.

We have implemented our ideas as a seamless extension to AspectJ—the most popular realisation of AOP concept for Java. The implementation is based on the AspectBench Compiler—an extensible workbench for experimenting with new language features.

Finally, we have tested our implementation on a number of non-trivial examples to illustrate the use of a purity-enabled compiler.

8.2 Future Work

Besides dynamic cross-cutting, AspectJ also provides means for static cross-cutting like introductions. The impact of having them in the aspect was out of scope of this work, but it may prove useful to investigate how they could affect aspect purity.

Purity verification algorithm presented in the dissertation is approximate. In particular, though sometimes possible, no termination analysis for pieces of advice is ever performed. Another source of inexactness is the fact that used points-to analysis engine is flow-insensitive, leading to possibility of false impurity detections. The overall accuracy can be improved by adding termination analysis and using more precise points-to information.

As mentioned in the corresponding chapter, **proceed()**-purity is not a necessary property of a pure aspect, if the **around()** advice replaces suppressed code with its exact equivalent. Though equivalence of two pieces of code cannot be always established, sometimes this is possible, and appropriate analysis could decrease the number of false impurity alerts due to **proceed()**-impurity.

An aspect declared as pure with respect to a certain module can, instead of directly modifying some value defined within that module, pass the reference to a third module. If the latter mutates the value sometime in the future, such purity violation will not be detected by our verifier, since the value in question is not modified in the control flow of any of the aspect's pieces of advice. Though this kind of thing does not happen in the most common usages of aspects, and such aspect would not be totally pure anyway (at the least, it would be detected as impure with respect to the module it had passed the reference to), nevertheless developing an algorithm for detection of such violations would make the analysis more sensitive. A flow-sensitive points-to information will be required for this algorithm.

Also a more extensive imperative study of benefits offered by purity annotations needs to be conducted. Such a study can discover possible new uses and shortcomings of the purity concept.

Bibliography

[1] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[2] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[3] Aspect-oriented software developement. `http://www.aosd.net/`.

[4] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[5] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[6] Aspectj home page. `http://www.aspectj.org/`.

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

[8] Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.

[9] Curtis Clifton and Gary T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science, October 2002.

[10] Definition of plug-in in wikipedia. `http://en.wikipedia.org/wiki/Plugin`.

[11] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.

[12] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, March 2004.

[13] Eiffel software home page. `http://www.eiffel.com/`.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[15] J. Aldrich. Open modules: Reconciling extensibility and information hiding, 2004.

[16] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.

[17] Benet Devereux. Compositional reasoning about aspects using alternating-time logic. In Leavens and Clifton [31].

[18] Henny B. Sipma. A formal model for cross-cutting modular transition systems. In Leavens and Clifton [31].

[19] Daniel S. Dantas and David Walker. Harmless advice. In *Workshop on Foundations of Object-Oriented Languages*, January 2005.

[20] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.

[21] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java, 2003.

[22] Polyglot extensible compiler framework. `http://www.cs.cornell.edu/Projects/polyglot/`.

[23] Ppg: Polyglot parser generator. `http://www.cs.cornell.edu/Projects/polyglot/ppg.html`.

[24] Cup lalr parser generator. `http://www2.cs.tum.edu/projects/cup/`.

[25] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[26] Soot: a java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[27] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[28] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, 1994. DIKU Research Report 94/19.

[29] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[30] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[31] Gary T. Leavens and Curtis Clifton, editors. *FOAL: Foundations of Aspect-Oriented Languages*, March 2003.

Appendix A

Purity Verifier Source Code

```
1  package abc.purity;
2
3  import java.util.*;
4
5  import polyglot.types.SemanticException;
6  import polyglot.util.ErrorInfo;
7  import polyglot.util.InternalCompilerError;
8  import polyglot.util.Position;
9
10 import abc.purity.weaving.aspectinfo.PureAspect;
11 import abc.weaving.aspectinfo.*;
12 import abc.weaving.matching.*;
13 import abc.weaving.residues.*;
14
15 import soot.*;
16 import soot.jimple.*;
17 import soot.tagkit.SourceFileTag;
18 import soot.tagkit.SourceLnPosTag;
19 import soot.util.Chain;
20
21 public class PurityVerifier extends SceneTransformer {
22
23     // hooks needed by Framwork:
24     private static PurityVerifier instance = new PurityVerifier();
25     private PurityVerifier() {}
26
27     public static PurityVerifier v() {
```

```
28          return instance;
29      }
30
31
32      private Map stmtShadowMatchLists = new HashMap();
33      private Map bodyShadowMatchLists = new HashMap();
34      private Map classesToPureAspects = new HashMap();
35
36      private Map modifiedByMethod = new HashMap();
37
38      protected void internalTransform(String phaseName, Map options) {
39          PointsToAnalysis p2a = Scene.v().getPointsToAnalysis();
40          Set methodsTakingParams/*<SootMethod>*/ = new HashSet();
41          Map paramLocals = new HashMap();
42          Map modifiedLocals = new HashMap();
43
44          // collect classes that represent pure aspects
45          Iterator aspectIt = GlobalAspectInfo.v().getAspects().iterator();
46          while (aspectIt.hasNext()) {
47              Aspect aspect = (Aspect)aspectIt.next();
48              if (aspect instanceof PureAspect)
49                  classesToPureAspects.put(
50                      aspect.getInstanceClass().getSootClass(), aspect);
51          }
52
53          // collect aspect methods taking params
54          Iterator classIt = Scene.v().getClasses().iterator();
55          while (classIt.hasNext()) {
56              SootClass sc = (SootClass)classIt.next();
57              if (sc.isApplicationClass()) {
58                  Iterator mIt = sc.getMethods().iterator();
59                  while (mIt.hasNext()) {
60                      SootMethod sm = (SootMethod)mIt.next();
61                      if (sm.hasActiveBody()) {
62                          if (isAround(sm.getName()))
63                              checkAround(sm);
64                          findMethodShadows(sm);
65                          Iterator unitIt =
66                              sm.getActiveBody().getUnits().iterator();
67                          while (unitIt.hasNext()) {
68                              Stmt stmt = (Stmt)unitIt.next();
69                              if (stmt.containsInvokeExpr()) {
```

```
70                                  SootClass invokedClass =
71                  stmt.getInvokeExpr().getMethod().getDeclaringClass();
72                                  if (invokedClass != sc &&
73                  classesToPureAspects.keySet().contains(invokedClass)) {
74                                      PureAspect invokedPureAspect =
75                  (PureAspect)classesToPureAspects.get(invokedClass);
76                  if (invokedPureAspect.getPureOn().matchesClass(sc) &&
77                      !excludeMethod(sm, invokedPureAspect))
78              methodsTakingParams.add(stmt.getInvokeExpr().getMethod());
79                                  }
80                              }
81                          }
82                      }
83                  }
84              }
85          }
86
87          // collect values collected by pure aspects
88          // collect values modified by pure aspects
89          aspectIt = GlobalAspectInfo.v().getAspects().iterator();
90          while (aspectIt.hasNext()) {
91              Aspect aspect = (Aspect)aspectIt.next();
92              if (aspect instanceof PureAspect) {
93                  SootClass sc = aspect.getInstanceClass().getSootClass();
94                  Iterator mIt = sc.getMethods().iterator();
95                  while (mIt.hasNext()) {
96                      SootMethod sm = (SootMethod)mIt.next();
97                      if (methodsTakingParams.contains(sm))
98                          collectParams(sm, paramLocals);
99                      exploreMethod(sm, sc);
100                     flattenSet((Set)modifiedByMethod.get(sm));
101                     Set modifiedLocalsSet = (Set)modifiedLocals.get(sc);
102                     if (modifiedLocalsSet == null) {
103                         modifiedLocalsSet = new HashSet();
104                         modifiedLocals.put(sc, modifiedLocalsSet);
105                     }
106                     modifiedLocalsSet.addAll(
107             getModificationsByMethod(sm, (PureAspect)aspect));
108                 }
109             }
110         }
111
```

```
112         // search for purity violations
113         Map purityViolations/*<SootClass, Set<Stmt>>*/ = new HashMap();
114
115         Iterator pureAspectClassesIt =
116             classesToPureAspects.keySet().iterator();
117         while (pureAspectClassesIt.hasNext()) {
118             SootClass pureAspectClass =
119             (SootClass)pureAspectClassesIt.next();
120             Set paramLocalsSet = (Set)paramLocals.get(pureAspectClass);
121             Set modifiedLocalsSet = (Set)modifiedLocals.get(pureAspectClass);
122             if ((paramLocalsSet != null) && (modifiedLocalsSet != null)) {
123                 Iterator modifiedLocalsIt = modifiedLocalsSet.iterator();
124                 while (modifiedLocalsIt.hasNext()) {
125                     Modification mod = (Modification)modifiedLocalsIt.next();
126                     Value v = mod.getValue();
127                     if (v instanceof StaticFieldRef) {
128                         PureAspect pureAspect =
129             (PureAspect)classesToPureAspects.get(pureAspectClass);
130                         StaticFieldRef sfr = (StaticFieldRef)v;
131                         if (
132         pureAspectClass != sfr.getField().getDeclaringClass()
133         && pureAspect.getPureOn().matchesClass(
134         sfr.getField().getDeclaringClass())) {
135                  Set violations = (Set)purityViolations.get(pureAspectClass);
136                          if (violations == null) {
137                              violations = new HashSet();
138                              purityViolations.put(pureAspectClass, violations);
139                          }
140                              violations.add(mod.getModifyingStmt());
141                          }
142                           continue;
143                       } else if (v instanceof Local) {
144                           Local ml = (Local)v;
145                           Iterator paramLocalsIt = paramLocalsSet.iterator();
146                           while (paramLocalsIt.hasNext()) {
147                             Local pl = (Local)paramLocalsIt.next();
148       if (p2a.reachingObjects(ml).hasNonEmptyIntersection(p2a.reachingObjects(pl))) {
149             Set violations = (Set)purityViolations.get(pureAspectClass);
150                     if (violations == null) {
151                          violations = new HashSet();
152                             purityViolations.put(pureAspectClass, violations);
153                                }
```

```
154                                 violations.add(mod.getModifyingStmt());
155                             }
156                         }
157                     }
158                 }
159             }
160         }
161
162         // report violations
163         Iterator violatingAspectsIt = purityViolations.keySet().iterator();
164         while (violatingAspectsIt.hasNext()) {
165             SootClass violatingAspect = (SootClass)violatingAspectsIt.next();
166             String fileName =
167     ((SourceFileTag)violatingAspect.getTag("SourceFileTag")).getSourceFile();
168             Set violations = (Set)purityViolations.get(violatingAspect);
169             Iterator violatingStmtsIt = violations.iterator();
170             while (violatingStmtsIt.hasNext()) {
171                 Stmt violatingStmt = (Stmt)violatingStmtsIt.next();
172                 SourceLnPosTag slpTag =
173         (SourceLnPosTag)violatingStmt.getTag("SourceLnPosTag");
174                 Position pos;
175                 if (slpTag != null)
176                     pos = new Position(fileName,
177     slpTag.startLn(), slpTag.startPos(), slpTag.endLn(), slpTag.endPos());
178                 else {
179                     pos = new Position(fileName);
180                 }
181                 abc.main.Main.v().error_queue.enqueue(
182             ErrorInfo.WARNING, "Aspect purity violated", pos);
183             }
184         }
185     }
186
187     private static void collectParams(SootMethod sm, Map paramLocals) {
188         Body body = sm.getActiveBody();
189         int paramCount = sm.getParameterCount();
190         Set locals = (Set)paramLocals.get(sm.getDeclaringClass());
191         if (locals == null) {
192             locals = new HashSet();
193             paramLocals.put(sm.getDeclaringClass(), locals);
194         }
195         for (int i = 0; i < paramCount; i++)
```

```
196             locals.add(body.getParameterLocal(i));
197     }
198
199     private void exploreMethod(SootMethod method, SootClass pureAspectClass) {
200         if (!method.getDeclaringClass().isApplicationClass())
201         //if (!method.hasActiveBody())
202             return;
203         if (modifiedByMethod.keySet().contains(method))
204             return;
205         modifiedByMethod.put(method, new HashSet());
206         Body body = method.getActiveBody();
207         Iterator unitIt = body.getUnits().iterator();
208         while (unitIt.hasNext()) {
209             Stmt stmt = (Stmt)unitIt.next();
210             if (!excludeStmt(method, stmt,
211         (PureAspect)classesToPureAspects.get(pureAspectClass)))
212               exploreStatement(method, stmt, pureAspectClass);
213         }
214     }
215
216     private void exploreStatement(SootMethod method, Stmt stmt,
217                             SootClass pureAspectClass) {
218         Set modified = (Set)modifiedByMethod.get(method);
219         if (stmt instanceof AssignStmt) {
220             AssignStmt assignStmt = (AssignStmt)stmt;
221             Value v = ((ValueBox)assignStmt.getDefBoxes().get(0)).getValue();
222             if (v instanceof InstanceFieldRef)
223                 while (v instanceof InstanceFieldRef) {
224                     v = ((InstanceFieldRef)v).getBase();
225                     modified.add(v);
226                 }
227             else if (v instanceof StaticFieldRef)
228                 modified.add(v);
229         }
230         if (stmt.containsInvokeExpr()) {
231             if (!excludeMethod(stmt.getInvokeExpr().getMethod(),
232                 (PureAspect)classesToPureAspects.get(pureAspectClass)))
233                 if (modifiedByMethod.containsKey(
234                     stmt.getInvokeExpr().getMethod()))
235                         modified.add(
236                             modifiedByMethod.get(
237                                 stmt.getInvokeExpr().getMethod()));
```

```
238                 else {
239                     exploreMethod(
240                         stmt.getInvokeExpr().getMethod(), pureAspectClass);
241                     Object o = modifiedByMethod.get(
242                             stmt.getInvokeExpr().getMethod());
243                     if (o != null) {
244                         Set md = (Set)o;
245                         if (!md.isEmpty()) {
246                             flattenSetBut(md, md);
247                             modified.addAll(md);
248                             modified.remove(modified);
249                         }
250                     }
251                 }
252         }
253     }
254 
255     private void flattenSet(Set set) {
256         Iterator sIt = set.iterator();
257         Set toRemove = new HashSet();
258         Set toAdd = new HashSet();
259         while (sIt.hasNext()) {
260             Object v = sIt.next();
261             if (v instanceof Set) {
262                 flattenSet((Set)v);
263                 toRemove.add(v);
264                 toAdd.addAll((Set)v);
265             }
266         }
267         set.removeAll(toRemove);
268         set.addAll(toAdd);
269     }
270 
271     private void flattenSetBut(Set set, Set but) {
272         Iterator sIt = set.iterator();
273         Set toRemove = new HashSet();
274         Set toAdd = new HashSet();
275         while (sIt.hasNext()) {
276             Object v = sIt.next();
277             if (v instanceof Set && v != but) {
278                 flattenSetBut((Set)v, but);
279                 toRemove.add(v);
```

```
280                 toAdd.addAll((Set)v);
281             }
282         }
283         set.removeAll(toRemove);
284         set.addAll(toAdd);
285     }
286 
287     private void flattenRemove(Set set, Set remove) {
288         set.remove(remove);
289         Iterator sIt = set.iterator();
290         Set rm = new HashSet();
291         while (sIt.hasNext()) {
292             Object v = sIt.next();
293             if (v instanceof Set)
294                 flattenRemove((Set)v, remove);
295         }
296     }
297 
298     private Set getModificationsByMethod(SootMethod sm, PureAspect aspect) {
299         Set result = new HashSet();
300         Body body = sm.getActiveBody();
301         Iterator unitIt = body.getUnits().iterator();
302         while (unitIt.hasNext()) {
303             Stmt stmt = (Stmt)unitIt.next();
304             if (!excludeStmt(sm, stmt, aspect)) {
305                 if (stmt instanceof AssignStmt) {
306                     AssignStmt assignStmt = (AssignStmt)stmt;
307                     Value v =
308                     ((ValueBox)assignStmt.getDefBoxes().get(0)).getValue();
309                     if (v instanceof InstanceFieldRef)
310                         while (v instanceof InstanceFieldRef) {
311                             v = ((InstanceFieldRef)v).getBase();
312                             result.add(new Modification(v, stmt));
313                         }
314                     else if (v instanceof StaticFieldRef)
315                         result.add(new Modification(v, stmt));
316                 }
317             }
318             if (stmt.containsInvokeExpr()
319       && modifiedByMethod.keySet().contains(stmt.getInvokeExpr().getMethod())) {
320                 Iterator it =
321         ((Set)modifiedByMethod.get(stmt.getInvokeExpr().getMethod())).iterator();
```

```
322                 while (it.hasNext())
323                     result.add(new Modification((Value)it.next(), stmt));
324             }
325         }
326         return result;
327     }
328
329     private void findMethodShadows(SootMethod sm) {
330         // whole body shadows
331         if(MethodCategory.weaveExecution(sm))
332             doShadows(sm, new WholeMethodPosition(sm));
333         // statement shadows
334         if(MethodCategory.weaveInside(sm)) {
335             Chain stmtsChain = sm.getActiveBody().getUnits();
336             Stmt current, next;
337
338             if(!stmtsChain.isEmpty()) {
339                 for(current =
340                 (Stmt)stmtsChain.getFirst(); current != null; current = next) {
341                     next = (Stmt)stmtsChain.getSuccOf(current);
342                     doShadows(sm, new StmtMethodPosition(sm, current));
343                     doShadows(sm, new NewStmtMethodPosition(sm, current, next));
344                 }
345             }
346         }
347
348         // exception handler shadows
349         Chain trapsChain = sm.getActiveBody().getTraps();
350         Trap currentTrap;
351
352         if(!trapsChain.isEmpty()) {
353             for(currentTrap = (Trap)trapsChain.getFirst();
354         currentTrap != null;
355         currentTrap=(Trap) trapsChain.getSuccOf(currentTrap))
356                 doShadows(sm, new TrapMethodPosition(sm, currentTrap));
357         }
358     }
359
360     private void doShadows(SootMethod method, MethodPosition pos) {
361         Iterator shadowTypeIt = abc.main.Main.v().getAbcExtension().shadowTypes();
362         while (shadowTypeIt.hasNext()) {
363             ShadowType st = (ShadowType)shadowTypeIt.next();
```

```
364             ShadowMatch sm;
365             try {
366                 sm = st.matchesAt(pos);
367             } catch(InternalCompilerError e) {
368                 throw new InternalCompilerError(e.message(),
369         e.position() == null ?
370         abc.polyglot.util.ErrorInfoFactory.getPosition(
371             pos.getContainer(),pos.getHost()) : e.position(), e.getCause());
372             } catch(Throwable e) {
373                 throw new InternalCompilerError
374                 ("Error while looking for join point shadow",
375                 abc.polyglot.util.ErrorInfoFactory.getPosition(
376                     pos.getContainer(),pos.getHost()), e);
377             }
378             if (sm instanceof StmtShadowMatch) {
379                 List list;
380                 list = (List)stmtShadowMatchLists.get(method);
381                 if (list == null) {
382                     list = new LinkedList();
383                     stmtShadowMatchLists.put(method, list);
384                 }
385                 list.add(sm);
386             } else if (sm instanceof BodyShadowMatch) {
387                 List list;
388                 list = (List)bodyShadowMatchLists.get(method);
389                 if (list == null) {
390                     list = new LinkedList();
391                     bodyShadowMatchLists.put(method, list);
392                 }
393                 list.add(sm);
394             }
395 
396         }
397     }
398 
399     private List getStmtShadowMatchList(SootMethod method) {
400         if(stmtShadowMatchLists.containsKey(method)) {
401             return (List)stmtShadowMatchLists.get(method);
402         } else {
403             return new LinkedList();
404         }
405     }
```

```
406
407     private List getBodyShadowMatchList(SootMethod method) {
408         if(bodyShadowMatchLists.containsKey(method)) {
409             return (List)bodyShadowMatchLists.get(method);
410         } else {
411             return new LinkedList();
412         }
413     }
414
415     public List getShadowMatchList(SootMethod method) {
416         LinkedList result = new LinkedList();
417         result.addAll(getStmtShadowMatchList(method));
418         result.addAll(getBodyShadowMatchList(method));
419         return result;
420     }
421
422     private boolean excludeStmt(SootMethod method, Stmt stmt, PureAspect aspect)
423     {
424         boolean result = false;
425         Pointcut pc = Pointcut.normalize(aspect.getExcept(),
426             new LinkedList(), aspect);
427         Iterator stmtShadowMatchIt = getStmtShadowMatchList(method).iterator();
428         while (stmtShadowMatchIt.hasNext()) {
429             StmtShadowMatch sm = (StmtShadowMatch)stmtShadowMatchIt.next();
430             Residue res = null;
431             try {
432                 res = pc.matchesAt(new EmptyFormals(),
433                     method.getDeclaringClass(), method, sm);
434             } catch (SemanticException se) {
435             }
436             if ((res instanceof AlwaysMatch) && (sm.getStmt() == stmt)) {
437                 result = true;
438                 break;
439             }
440         }
441         return result;
442     }
443
444     private boolean excludeMethod(SootMethod method, PureAspect aspect) {
445         boolean result = false;
446         Pointcut pc = Pointcut.normalize(aspect.getExcept(),
447                         new LinkedList(), aspect);
```

```
448         Iterator bodyShadowMatchIt = getBodyShadowMatchList(method).iterator();
449         while (bodyShadowMatchIt.hasNext()) {
450             BodyShadowMatch sm = (BodyShadowMatch)bodyShadowMatchIt.next();
451             Residue res =    null;
452             try {
453                 res = pc.matchesAt(new EmptyFormals(),
454                         method.getDeclaringClass(), method, sm);
455             } catch (SemanticException se) {
456             }
457             if (res instanceof AlwaysMatch) {
458                 result = true;
459                 break;
460             }
461         }
462         return result;
463     }
464
465     private boolean isAround(String methodName) {
466         boolean result = false;
467         if (methodName.startsWith("around$"))
468             result = true;
469         return result;
470     }
471
472     private boolean isProceed(String methodName) {
473         boolean result = false;
474         if (methodName.startsWith("abc$static$proceed$"))
475             result = true;
476         return result;
477     }
478
479     private void checkAround(SootMethod method) {
480         Chain units = method.getActiveBody().getUnits();
481         Local[] locals = new Local[method.getParameterCount()];
482         for (int i = 0; i < locals.length; i++)
483             locals[i] = method.getActiveBody().getParameterLocal(i);
484         Stmt first = (Stmt)units.getFirst();
485         if (checkNumProceeds(units, first, new HashMap(), locals)[1] != 1) {
486             String fileName =
487         ((SourceFileTag)method.getDeclaringClass().getTag(
488                 "SourceFileTag")).getSourceFile();
489             SourceLnPosTag slpTag =
```

```
490                 (SourceLnPosTag)method.getTag("SourceLnPosTag");
491             Position pos = new
492                 Position(fileName, slpTag.startLn(),
493                         slpTag.startPos(),
494                         slpTag.endLn(),
495                         slpTag.endPos());
496             abc.main.Main.v().error_queue.enqueue(ErrorInfo.WARNING,
497             "Possibly proceed()-purity violated in around() advice", pos);
498         }
499     }
500
501     private int[] checkNumProceeds(Chain units, Stmt start,
502             Map ifsAlreadySeen, Local[] locals) {
503         boolean finished = false;
504         int[] numProceeds = {0, 0};
505         Stmt current = start;
506         while (numProceeds[0] < 2 && numProceeds[0] == numProceeds[1]
507                 && !finished && current != null) {
508             if (current.containsInvokeExpr()) {
509                 if (isProceed(current.getInvokeExpr().getMethod().getName())) {
510                     numProceeds[0]++;
511                     boolean ok = true;
512                     List args = current.getInvokeExpr().getArgs();
513                     for (int i = 0; i < args.size(); i++)
514                         if (args.get(i) != locals[i]) {
515                             ok = false;
516                             break;
517                         }
518                     if (ok)
519                         numProceeds[1]++;
520                 }
521                 current = (Stmt)units.getSuccOf(current);
522             } else if (current instanceof TableSwitchStmt) {
523                 TableSwitchStmt tsStmt = (TableSwitchStmt)current;
524                 List targets = tsStmt.getTargets();
525                 int maxBranchProceeds[] = {0, 0};
526                 for (int i = 0; i < targets.size(); i++) {
527                     int[] branchProceeds = checkNumProceeds(units,
528                     (Stmt)targets.get(i), ifsAlreadySeen, locals);
529                     if (branchProceeds[0] > maxBranchProceeds[0])
530                         maxBranchProceeds[0] = branchProceeds[0];
531                     if (branchProceeds[1] > maxBranchProceeds[1])
```

```
532                         maxBranchProceeds[1] = branchProceeds[1];
533                 }
534                 numProceeds[0] += maxBranchProceeds[0];
535                 numProceeds[1] += maxBranchProceeds[1];
536                 finished = true;
537             } else if (current instanceof LookupSwitchStmt) {
538                 LookupSwitchStmt lsStmt = (LookupSwitchStmt)current;
539                 List targets = lsStmt.getTargets();
540                 int[] maxBranchProceeds = {0, 0};
541                 for (int i = 0; i < targets.size(); i++) {
542                     int[] branchProceeds = checkNumProceeds(units,
543                         (Stmt)targets.get(i), ifsAlreadySeen, locals);
544                     if (branchProceeds[0] > maxBranchProceeds[0])
545                         maxBranchProceeds[0] = branchProceeds[0];
546                     if (branchProceeds[1] > maxBranchProceeds[1])
547                         maxBranchProceeds[1] = branchProceeds[1];
548                 }
549                 numProceeds[0] += maxBranchProceeds[0];
550                 numProceeds[1] += maxBranchProceeds[1];
551                 finished = true;
552             } else if (current instanceof GotoStmt) {
553                 int[] nextProceeds = checkNumProceeds(units,
554             ((Stmt)((GotoStmt)current).getTarget()), ifsAlreadySeen, locals);
555                 numProceeds[0] += nextProceeds[0];
556                 numProceeds[1] += nextProceeds[1];
557                 finished = true;
558             } else if (current instanceof IfStmt) {
559                 if (!ifsAlreadySeen.keySet().contains(current)) {
560                     ifsAlreadySeen.put(current, new Integer(1));
561                     int[] nextProceeds1 =
562                             checkNumProceeds(units,
563                             (Stmt)units.getSuccOf(current),
564                             ifsAlreadySeen, locals);
565                     int[] nextProceeds2 = checkNumProceeds(units,
566                             ((IfStmt)current).getTarget(),
567                                 ifsAlreadySeen, locals);
568                 numProceeds[0] += Math.max(nextProceeds1[0], nextProceeds2[0]);
569                 numProceeds[1] += Math.max(nextProceeds1[1], nextProceeds2[1]);
570                 } else {
571                     int n = ((Integer)ifsAlreadySeen.get(current)).intValue();
572                     if (n < 2) {
573                         ifsAlreadySeen.put(current, new Integer(++n));
```

```
574                         int[] nextProceeds = checkNumProceeds(units,
575                             ((IfStmt)current).getTarget(),
576                                 ifsAlreadySeen, locals);
577                         numProceeds[0] += nextProceeds[0];
578                         numProceeds[1] += nextProceeds[1];
579                     }
580                 }
581                 finished = true;
582             } else if (current instanceof ReturnStmt
583                     || current instanceof ReturnVoidStmt) {
584                 finished = true;
585             } else
586                 current = (Stmt)units.getSuccOf(current);
587 
588         }
589         return numProceeds;
590     }
591 
592     private static class Modification {
593         private Value value;
594         private Stmt modifyingStmt;
595 
596         public Modification(Value value, Stmt modifyingStmt) {
597             this.value = value;
598             this.modifyingStmt = modifyingStmt;
599         }
600 
601         public Value getValue() {
602             return value;
603         }
604 
605         public Stmt getModifyingStmt() {
606             return modifyingStmt;
607         }
608     }
609 }
```