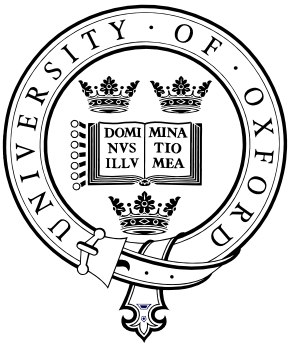
Efficient Implementation of Around-Advice for the AspectBench Compiler



Oxford University

Computing Laboratory

Sascha Kuzins
Lady Margaret Hall

September 2, 2004

Abstract

In Aspect Oriented Programming, aspects observe a base program and execute extra code – the advice – when certain patterns of events occur. Advice can be executed before, after of instead of the original event. *Around*-advice executes instead of the original code of the base program, but is allowed to explicitly invoke the original code.

The most popular implementation language for aspects is AspectJ, an extension of Java.

This dissertation presents an efficient and complete code generation strategy for *around* advice as part of the AspectBench Compiler `abc`. Measurements are presented which show that the strategy can yield perfomance gains of 200% or more when compared to the official AspectJ compiler `ajc`.

Acknowledgements

I would like to thank Professor Oege de Moor for letting me work on this project and for providing support and guidance throughout. Many thanks to Professor Laurie Hendren, who was of great help during the initial phase of the project.

It has been a great experience to work on the AspectBench Compiler team, and I would like to thank everyone for providing such an interesting and fun environment.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Aspect Oriented Programming (AOP) is a new programming paradigm that addresses cross-cutting concerns. Cross-cutting concerns are concerns that cross-cut traditional boundaries of abstraction. The aim of AOP is to allow the programmer to cleanly express cross-cutting concerns in a modular way.

The most commonly cited example for the use of AOP is logging of events. AOP allows the programmer to add such logging functionality to a program without modifying the original program code. To achieve this, the programmer writes an aspect that is applied to the base program to add the logging functionality.

The design of language features for catching deviant behaviour at runtime is highly worthwhile: for example, the current codebase for Microsoft Word contains over 3000 macros for logging purposes.

1.1 Contribution of Thesis

The aim of this work is to provide an efficient implementation of the most versatile kind of advice of the language, namely *around* advice.

The contributions of this work are

- to provide a novel weaving strategy that generally neither relies on inlining nor closures
- to show an implementation strategy for multiple variable bindings in conjunction with around advice

As we shall demonstrate, our implementation strategy results in massive speedups over the state-of-the-art.

1.2 Thesis Organization

Chapter 2 starts out with an overview of Aspect Oriented Programming and some simple examples. The chapter gives a brief introduction to the AspectJ language with a focus on *around* advice. The chapter concludes with a section on the AspectBench Compiler `abc`, its architecture and components.

Chapter 3 describes the actual code generation strategy of the around weaver presented in this thesis. The strategy is illustrated using example AspectJ programs together with the code that is generated by the weaver. Each feature is described separately, and the aim was to reduce the example programs as far as possible to clearly demonstrate the essentials of each feature. The appendix of this thesis contains the decompilation of a more complex example.

Chapter 4 focuses on the implementation of the weaver, including its structure and class and object hierarchies.

Chapter 5 shows benchmarks of programs generated by `abc` and compares the results with programs generated by the official AspectJ compiler `ajc`.

Chapter 6 summarizes the results of this work and shows potential future additions and optimizations.

Chapter 2

Background

2.1 Aspect Oriented Programming

Aspect Oriented Programming allows the programmer to write *aspects* that monitor and modify the behaviour of a base program without modifying the source code of the base program itself.

Conceptually, aspects observe the base program at run-time. This is in contrast to meta-programming techniques, which perform transformations at compile time. An efficient implementation will try to do static analyses at compile time to minimize the overhead at run-time.

A *dynamic joinpoint* is an event in the program. The *joinpoint shadow* is the program point that corresponds to the joinpoint. A *pointcut* defines a pattern of events or joinpoints. *Advice* is extra code that is executed.

Intertype Declarations are part of the static part of the language and allow the introduction of new members into existing classes at compile time.

We shall now illustrate these definitions with a number of example programs before proceeding to a more detailed description.

2.1.1 Logging

Logging is a common debugging technique. Traditionally, the programmer would have to insert calls to the logging facility at every point of interest in the program. This is undesirable because this one concern is spread throughout the whole program.

In AspectJ, the concern of logging can be implemented in a single aspect, thus preserving modularity. The following aspect implements a very simple logger that logs all dynamic joinpoints of the base program.

Listing 2.1: Logging aspect

```
1 aspect Logger
2 {
3     before() : !within(Logger) {
4         System.out.println(thisJoinPoint);
5     }
6 }
```

The aspect contains one piece of *before* advice. This advice is executed before the execution of the joinpoint. The poincut expression !**within**(Logger) picks out all joinpoints that are not within the aspect itself. The advice body simply prints out information about the current joinpoint. It uses the special keyword **thisJoinPoint**. This keyword returns an interface through which information about the joinpoint can be queried.

When the aspect is applied to the following base program, the output in Figure 2.1 is produced.

Listing 2.2: Simple base program

```
1 public class Foo
2 {
3     public static void main(String args[])
4     {
5         bar();
6     }
7     public static void bar() {}
8 }
```

```
staticinitialization(Foo.<clinit>)
execution(void Foo.main(String[]))
call(void Foo.bar())
execution(void Foo.bar())
```

Figure 2.1: Logger output

This illustrates some of the joinpoints that are supported by AspectJ: one can intercept the static initialisation of a class (here Foo), the execution of a method (here main), and the call of a method (here Foo.bar()). The distinction between **call** and **execution** is that between the call site and the method body. This example only shows three of the joinpoint types that AspectJ supports; there are 11 in all, and we shall present a full list in Section 2.2.1.

2.1.2 Enforcing Coding Standards

As described in [3], programmers sometimes signal error conditions by returning **null** from methods. This can be regarded as a poor programming

style since the exception mechanism provides a more structured way of signalling errors.

The *nullcheck* example shows how aspects can be used to enforce such coding standards at runtime.

The aspect below monitors the base program and logs violations of the coding convention. The originator of this example chose to implement it using *around* advice.

The pointcut expression methodsThatReturnObjects matches all calls to methods that return objects. To wit, the pointcut (**call**(Object+ *.*(..))) describes calls to methods in any class (the first star) with any name (the second star) and any number of parameters (the (..)), whose return type is a subtype of Object (the subtype matching is indicated by the +). The *around* advice is executed for all these calls. The advice uses the special **proceed** statement to proceed with the original method call. It then checks the returned value and signals the error condition if appropriate.

Interestingly, the pointcut in this aspect matches two joinpoints within the advice itself, namely the call to **thisJoinPoint**.getSignature() and the implicit StringBuffer.append() that the compiler generates for the + operator. As observed in [3], this form of self referencing advice causes the official AspectJ compiler `ajc` to use its closure strategy, which results in a slow execution of the resulting program.

Listing 2.3: Nullcheck aspect

```
1  aspect Aspect
2  {
3      pointcut methodsThatReturnObjects():
4          call(Object+ *.*(..));
5
6      Object around(): methodsThatReturnObjects()
7      {
8          Object lRetVal = proceed();
9          if(lRetVal == null)
10         {
11             System.err.println(
12                 "Null return value: " +
13                 thisJoinPoint.getSignature());
14         }
15         return lRetVal;
16     }
17 }
```

The following simple base program intentionally violates the coding convention.

Listing 2.4: Nullcheck example base program

```
1  public class Foo
2  {
3      public static void main(String args[])
4      {
5          bar();
6      }
7      public static Object bar() {
8          return null; // violates coding standard
9      }
10 }
```

The aspect produces the following output:

```
Null return value: Object Foo.bar()
```

Seasoned AspectJ programmers may object that the above example could have been more efficiently expressed using the **after returning** construct. Our reply is that the stark efficiency tradeoff between **around** and **after returning** is merely an artifact of the implementation strategy chosen by `ajc`, and indeed we intend to eliminate that peculiarity. Furthermore, the code shown here was directly taken from an on-line article by Dale Asberry (see [3]).

2.1.3 Caching

Caching is a well known method to speed up programs. This can be regarded as a cross-cutting concern.

- The concern of the base program is to produce the correct output.
- The concern of the caching is to speed up the program at the cost of runtime memory usage without modifying its external behaviour as a function.

Both concerns are independent of each other.

Using AOP, the caching functionality can be expressed in an aspect as a separate module.

As an example, let us consider the factorial function.

Listing 2.5: Factorial example

```
1  class Factorial {
2      public static long factorial(int n) {
3          return n==0 ? 1 : factorial(n-1) * n;
4      }
5  }
```

This function has linear complexity. Caching cannot speed up the initial invocation of the function. However, this function can especially benefit from caching if it is invoked multiple times.

The following aspect implements the caching concern. It contains two pieces of advice. The *around* advice intercepts every invocation of the function, checks the cache and on success returns the cached value. The *after returning* advice only intercepts the non-recursive factorial invocations using the **cflowbelow** statement and stores the results in the cache. This strategy prevents caching of all steps of the recursive calculation and thus reduces memory consumption.

Listing 2.6: Caching aspect

```
1  aspect FactorialCache {
2      private java.util.Map cache=new java.util.HashMap();
3
4      // named pointcut matching the execution
5      // of the factorial method
6      pointcut factorialMethod(int n):
7          execution(long Factorial.factorial (int)) &&
8          args(n);
9
10     after (int n) returning(long l):
11         factorialMethod(n) &&
12         !cflowbelow(factorialMethod(int)) // exclude
13                                           // recursive
14                                           // calls
15     {
16         // add the returned value to the cache
17         cache.put(new Integer(n), new Long(l));
18     }
19
20     long around(int n) : factorialMethod(n)
21     {
22         Integer lookupValue=new Integer(n);
23         Object cachedValue=cache.get(lookupValue);
24         if (cachedValue!=null)
25             // return cached value
26             return ((Long)cachedValue).longValue();
27         else
28             // cache miss: proceed with calculation.
29             return proceed(n);
30     }
31 }
```

2.2 The AspectJ Language

AspectJ[2] was originally developed at Xerox Palo Alto Research Center (PARC). It has now become part of the Eclipse project.

AspectJ is implemented as an extension to the Java programming language. AspectJ is intended to be a superset of Java, and the AspectJ compiler `ajc` should compile all valid Java programs. To achieve this, most AspectJ keywords are only recognized inside of aspects. Furthermore, the produced programs should run on the standard Java Virtual Machine (JVM).

The AspectJ compiler itself is released as open source under the Common Public License (CPL); is is maintained by a team of professional programmers at IBM.

`ajc` is a stand-alone compiler and is built on top of the Eclipse Java compiler.

Recall that an aspect observes the execution of the base program, executing advice when it finds joinpoints that match a given pointcut. To implement this in Java bytecode, `ajc` attempts to do the pointcut matching at compile time. Each shadow point is matched against the pointcut. If execution of the shadow might give rise to a matching joinpoint at runtime, the compiler inserts a dynamic test (the dynamic residue), and a call to the relevant advice. This process is called *weaving*.

`ajc` allows weaving into Java sources as well as Java class files. The output always consists of class files.

AspectJ comes with a runtime library that must be present when executing compiled programs. The runtime library contains interfaces that can be used in AspectJ programs. These are in the package `org.aspectj.lang`.

It follows a discussion of the concepts of AspectJ that are relevant for this thesis.

2.2.1 Joinpoints

Joinpoints are well defined events in the execution of a program. These events can trigger the execution of additional code.

Joinpoints are largely dynamic in nature: They can depend on the current call stack, or can be picked out using arbitrarily complex Java expressions.

Table 2.1 lists the joinpoints that are available in the AspectJ language [6] and describes the context values which they expose.

A *joinpoint shadow* is the static region of code that corresponds to a joinpoint. For example, the shadow corresponding to the execution of a method is the body of that method.

Joinpoint	**Exposed context**
Method call	Caller, target object, method arguments, returned value
Method execution	Instance, method arguments, returned value
Constructor call	Caller, constructor arguments
Constructor execution	Constructed object, constructor arguments
Static initializer execution	none
Field get	Referencing instance, target object, field value
Field set	Setting instance, target object, set value
Advice execution	Aspect, advice arguments, advice return value
Object pre-initialization	Constructor arguments
Object initialization	Instance, constructor arguments
Handler execution	Instance, exception

Table 2.1: Joinpoints in AspectJ

2.2.2 Pointcuts

A pointcut is a set of joinpoints. The AspectJ language contains a poincut language which is used to define pointcuts.

Pointcut expressions are built from pointcut primitives which can be combined using pointcut operators. Table 2.2 lists the available primitives. The operators are logical *and* (&&), logical *or* (||) and logical *not* (!).

A pointcut expression (or pointcut designator, PCD) picks out dynamic joinpoints. For each joinpoint, a PCD returns a set of bindings between advice formals and context values of the joinpoint[11].

2.2.3 Advice

Advice is extra code defined by the programmer. There are three different kinds of advice.

- *before* advice is executed before the execution of the joinpoint
- *after* advice is executed after the execution of the joinpoint. There are two specializations of *after* advice:
  - *after returning* advice is triggered if the joinpoint returns without throwing an exception. *after returning* advice can capture the return value of the joinpoint.
  - *after throwing* advice is triggered if the joinpoint throws an exception and can capture the thrown exception.

Pointcut primitive	Matched joinpoints
execution	Method and constructor execution
call	Method and constructor calls
staticinitialization	Static initializer execution
get	Field gets
set	Field sets
handler	Exception handlers
initialization	Object initialization, part after **super** call
preinitialization	Object initialization, part before **super** call
adviceexecution	execution of advice methods
cflow	joinpoints in the control flow of the joinpoints of a specified pointcut
cflowbelow	Equivalent to **cflow**, excluding the joinpoints of the specified pointcut
within	Joinpoints within specified classes
withincode	Joinpoints within specified methods
this	Joinpoints with a **this** context value of a specified type. Can bind the context value.
target	Joinpoints with a **target** context value of a specified type. Can bind the context value.
args	Joinpoints with arguments of specified types. Can bind the context values.
if	Joinpoints for which a boolean AspectJ expression evaluates to **true**

Table 2.2: Pointcut primitives

- *around* advice is executed instead of the joinpoint. *around* advice can then optionally invoke the original joinpoint. This thesis focuses on this type of advice, and the semantics of around advice are explained in greater detail in the next section.

2.2.4 Intertype declarations

This feature of AspectJ allows modifications of classes at compile time. Intertype declarations are purely static. In particular, intertype declarations support

- the addition of methods and member variables to other classes.
- modifications of the class hierarchy
- the declaration of compile time warnings and errors.

Intertype declarations are relatively independent of the dynamic part of the language and are not the focus of this thesis.

2.2.5 Aspects

Pointcuts, advice and intertype declarations are defined inside of *aspects*. Aspects are similar to classes, and additional methods, fields, initializers and classes can be defined within aspects.

Aspects are never directly instantiated by the programmer. Instead, aspects are created automatically, usually as a singleton instance or, as an advanced feature, on a per object basis.

2.3 Around-Advice

This section focuses on the syntax and semantics of around advice in AspectJ.

Around-advice surrounds the joinpoint's execution. Around-advice can bypass the execution, invoke the original execution and alter the context of the execution. [6]

Around-advice is the most general form of advice. All other kinds of advice, namely *before*, *after*, *after returning* and *after throwing*, can be transformed into around-advice (modulo some subtle issues in the type-checking rules, which are beyond the scope of this thesis).

Advanced examples of *around* advice usage are presented towards the end of this section.

2.3.1 Proceed

Within the advice body, the statement **proceed** invokes the original joinpoint.

proceed must be called like a method with the same signature as the around advice. The meaning of the arguments is discussed below in conjunction with *advice formals*.

The use of **proceed** is optional. **proceed** statements can occur in the body an arbitrary number of times, and each call can be invoked an arbitrary number of times.

The return type of the **proceed** call is the same as the return type of the around advice.

Local and anonymous classes

proceed can be called from within local or anonymous classes inside the around body.

One implication is that such **proceed** invocations can occur at an arbitrary time after the control flow of the program has left the advice body. Readers who are familiar with compiler construction will recognize that this points to closures as the obvious implementation strategy for **proceed**.

2.3.2 Advice Formals

Around advice can have arguments. Arguments are used to access the exposed context values of the joinpoint. The mapping of the arguments to context values is called *binding*. All of the advice formals must be bound to values of the current context. Binding is done in the pointcut using the pointcut primitives **args**, **this** and **target**.

The number and type of arguments that are passed to **proceed** must match the declaration of the around advice. The programmer can modify the context values of the joinpoint by passing different values to **proceed**.

The following simple example illustates this mechanism. The advice captures calls to all **void** methods that have one argument of type Integer. The *advice formal* n is bound to the argument of the call using **args**(n). This value is replaced by the value passed to **proceed**. In this example, **null** values are replaced by an Integer instance with the value 0, and all other values are passed on without modification.

Listing 2.7: Binding advice formals

```
1 void around(Integer n) : call(void *.*(Integer)) && args(n)
2 {
3     proceed(n==null ? new Integer(0) : n);
4 }
```

One context value can be bound to multiple advice arguments. In the advice declaration below, both x and y are bound to the **this** context value of the joinpoint.

If one context value is bound to multiple arguments and **proceed** is passed a different value for each argument, it is not obvious which value should eventually be assigned to the context value.

```
1 void around(Object x, Object y) : this(x) && this(y)
2 {
3     proceed(foo, bar); // foo and bar defined elsewhere
4 }
```

In this example, when executing the joinpoint, the question is whether the **this** value is assigned foo or bar.

The AspectJ programming guide does not specify this. Experiments with `ajc` show that in case of a conflict, the last parameter that is bound to the context value is chosen. The weaver presented in this thesis follows that convention.

Another special case arises if a pointcut expression contains multiple binding sub-expressions that bind different context values to the same advice argument. The effective binding may depend on dynamic checks and thus may only be known at runtime.

```
1 void around(Foo x) : args(x,...) || args(..,x)
2 ...
```

This pointcut matches a joinpoint if the first argument is of type Foo and binds x to this argument. The pointcut also matches if the first argument is *not* of type Foo, but the last argument is. In this case, x is bound to the last argument.

When passing a value to **proceed**, the question is whether to assign this value to the first argument or to the last argument. The only reasonable behaviour is to consider the effective binding at runtime and, in the case of passing changed values to **proceed**, modify the effectively bound context value.

This case of multiple bindings causes certain implementation difficulties. The `ajc` compiler (version 1.2) is limited in that it does not allow such bindings and aborts with a compiler limitation error. The around weaver presented in this thesis is able to correctly deal with this situation.

2.3.3 Return type and value

Around advice must have a return type (which can be **void**). The value returned by the advice replaces the original value of the joinpoint.

Object return type This type is treated differently from other types. [1] Advice with Object return type can apply to *any* joinpoint, regardless of its type.

- All reference types are of type Object and require no special treatment
- Simple types are automatically boxed and unboxed
- If the advised joinpoint has **void** type, the returned value is ignored. **proceed** returns **null** in this case.

2.3.4 Advanced Around-Advice Examples

We now present two more advanced examples for around advice. Each of these examples has a special way of using around. The examples are loosely based on ideas from [6].

[1] The abc team largely considers this special treatment an unfortunate design choice for the AspectJ language. The proposed alternative would be a syntax that explicitly indicates the special behaviour.

Thread Safety

This example shows how to use AspectJ to transparently enforce thread safety. The example uses the Swing Java GUI library. Since the library is not thread safe, the safety rule for this library is that only the event-dispatching thread should directly access Swing components. Other threads can access Swing components indirectly by calling EventQueue.invokeLater() or EventQueue.invokeAndWait() and passing these methods an object of type Runnable. The event-dispatching thread then executes the run() method of the object.

This approach has some disadvantages.

- It is error-prone since the programmer has to remember to use this indirect access method. The compiler will not issue any warnings if the safety rule is violated.
- Wrapping all the modifying code in Runnable classes clutters the source code and makes it less comprehensible.

Using AspectJ, the thread safety rule can be implemented transparently. The following aspect shows the implementation. [2]

```
1  aspect SwingThreadSafety {
2      pointcut uiMethodCalls() :
3          call(void javax..JComponent+.*(..))   ||
4          ... // more relevant packages
5
6      void around() : uiMethodCalls()
7      {
8          Runnable worker = new Runnable() {
9              public void run() {
10                 proceed();
11             }
12         };
13         EventQueue.invokeLater(worker);
14     }
15 }
```

The pointcut uiMethodCalls captures all calls to **void** Swing methods that have to obey the thread rule. The around advice is invoked for all of these calls. It creates the required Runnable object and adds it to the event queue.

In this example, **proceed** is used inside an anonymous class within the around advice body. It is noteworthy that the **proceed** call is invoked

- by a different thread

[2] To allow the reader to focus on the essential parts, the aspect has been simplified as far as possible and in this form only supports calls to void methods. A more detailed implementation can be found in [6].

- at a later point of time in the program's execution.

From an implementation point of view, in particular this means that the context of the joinpoint has to be kept until that invocation occurs.

Thread Pooling

In this example, the base program is a simple multi-threaded TCP server application. The server listens for incoming connections and for each incoming request creates a new handler thread that executes the request.

```
1  import java.util.*;
2  import java.io.*;
3  import java.net.*;
4
5  public class Server
6  {
7      public static final int PORT_NUM=0xABC;
8      public static void main(String[] args) throws Exception
9      {
10         ServerSocket serverSocket=
11             new ServerSocket(PORT_NUM);
12         while (true)
13         {
14             Socket requestSocket=serverSocket.accept();
15             Thread requestThread=
16                 new RequestHandler(requestSocket);
17             requestThread.start();
18         }
19     }
20 }
```

The handler thread has a method run() that performs the service to the client. The example handler simply sends a string to the client:

```
1  class RequestHandler extends Thread
2  {
3      private Socket requestSocket;
4      public RequestHandler(Socket requestSocket)
5      {
6          this.requestSocket=requestSocket;
7      }
8      public void run()
9      {
10         // process request ...
11         PrintWriter writer=null;
12         try {
13             writer=new PrintWriter(
14                     requestSocket.getOutputStream());
15             // simulate lengthy operation
```

```
16              try { Thread.sleep(1000); }
17              catch(InterruptedException e){}
18              writer.write("Hello!\n");
19              writer.flush();
20          } catch (IOException ex) {
21          } finally { // free resources
22              try {
23                  if (writer!=null)
24                      writer.close();
25                  requestSocket.close();
26              } catch (IOException ex) { }
27          }
28      }
29  }
```

In real world applications where performance is important, thread pooling is often used to lower the performance impact of thread creation and destruction.

Thread pooling can be regarded as a concern independent of the server functionality itself. Using AspectJ, thread pooling can be implemented in a separate module.

The following aspect shows such an implementation for the above base program. [3]

The aspect has three pieces of around advice. Their function is to

- modify the execution of the request handler to be able to run multiple times
- intercept the creation of new request handlers and to return an existing handler from the pool if possible
- intercept calls to the thread-start function and modify them to wake up the pooled thread if necessary.

The first piece of *around* advice is shown below. After every execution of the handler (which is invoked by **proceed**), the thread adds itself to the pool and waits.

```
1  privileged aspect ThreadPooling {
2      Set pool=new HashSet();
3
4      // Capture execution of request handler
5      void around(RequestHandler requestHandler) :
6          execution(void RequestHandler.run()) &&
7          this(requestHandler){
8          while (true) {
```

[3] The example has been simplified as far as possible for added clarity. For a more heavy-weight implementation, see [6].

```
9               // execute handler
10              proceed(requestHandler);
11              requestHandler.requestSocket=null;
12              // add handler to pool
13              synchronized(pool) {
14                  pool.add(requestHandler);
15              }
16              // wait
17              synchronized(requestHandler) {
18                  while (requestHandler.requestSocket==null) {
19                      try{ requestHandler.wait(); }
20                      catch   (InterruptedException e) {}
21                  }
22              }
23          }
24      }
25      ...
26  }
```

For every creation of a new request handler, the second piece of advice checks the pool for available threads. If such a thread exists, it is removed from the pool and returned. Otherwise, the original joinpoint is invoked which creates a new thread.

```
1  privileged aspect ThreadPooling {
2      ...
3      // intercept new handler creation
4      RequestHandler around(Socket socket) :
5          call(RequestHandler.new(Socket)) &&
6          args(socket) {
7          synchronized(pool) {
8              if (pool.isEmpty()) {
9                  // proceed with creation
10                 RequestHandler result=proceed(socket);
11                 return result;
12             } else {
13                 // recycle handler from pool
14                 RequestHandler  result=
15                     (RequestHandler)pool.iterator().next();
16                 pool.remove(result);
17                 result.requestSocket=socket;
18                 return result;
19             }
20         }
21     }
22     ...
23 }
```

Finally, the Thread.start() method has to be intercepted. Threads from the pool have already been started, so the advice notifies these threads to

resume execution instead.

```
1  privileged aspect ThreadPooling {
2      ...
3      // Intercept thread start
4      void around(RequestHandler requestHandler) :
5          call(void Thread.start()) &&
6          target(requestHandler) {
7          if (requestHandler.isAlive())
8              // if the thread is a reused thread,
9              // wake it up
10             synchronized(requestHandler)     {
11                 requestHandler.notify();
12             }
13         else
14             // for new threads,
15             // proceed with Thread.start()
16             proceed(requestHandler);
17     }
18 }
```

This example is special in that the around advice contains a **proceed** statement inside a loop. The **proceed** statement executes the body of the advised method repeatedly.

2.4 The AspectBench Compiler

2.4.1 Aims

The AspectBench Compiler (`abc`) [1] is an alternative compiler for the AspectJ language. `abc` is developed at the Oxford University Computer Laboratory by the Programming Tools Group in cooperation with the Sable Research Group at McGill University, Canada.

The need for a second AspectJ compiler arises out of some deficiencies of the original compiler `ajc`. These deficiencies led to the design goals of `abc`. The main design goals of `abc` are efficiency of the produced code, extensibility of the compiler and a clean design and language specification.

As of writing, the first version of `abc` is almost completed and these goals have largely been achieved.

The success of the `abc` project is in part due to its foundation. `abc` is based on a powerful optimization framework with an elegant intermediate representation of Java bytecode. The front-end of the compiler is based on an extensible Java front-end framework.

Incremental compilation, that is the support for separate compilation of the units of a program, is not supported by `abc`. Incremental compilation contradicts the design goals of producing efficient code and of a clean architectural design of the compiler.

2.4.2 Architecture

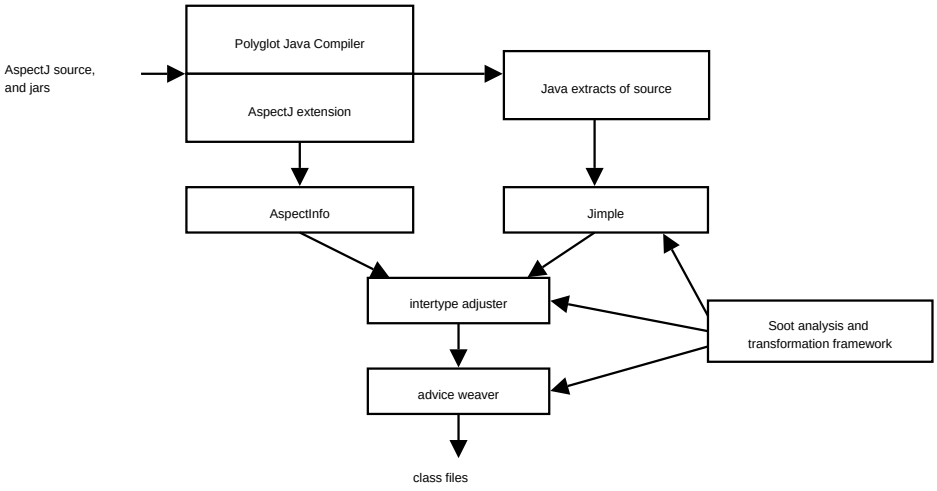


Figure 2.2: Architecture of abc

`abc` utilizes the Soot Optimization Framework [10], which in turn uses the Polyglot extensible Java compiler.

Polyglot

Polyglot [9] is an extensible compiler framework for Java. It was designed to create compilers for languages similar to Java and aims to avoid the need for code duplication.

Polyglot is written in Java and is open source.

The Soot Optimization Framework

Soot [10] is a framework for analyzing and transforming Java bytecode. It was developed at McGill University, Montreal. Soot itself is written in Java and is based on the GNU LGPL.

Soot offers multiple intermediate representations of Java bytecode (see Table 2.3).

Only *Jimple* is needed for our purposes.

Soot can read and write Java class files and transform between bytecode and the intermediate representations.

In addition, Soot uses Polyglot to read Java source files directly. Soot can also output Java source code and hence can be used as a Java decompiler.

Jimple

Jimple is a typed, stackless 3-address representation of the Java bytecode[10]. It is the principal representation in the soot framework.

Representation	Description
Baf	Compact, stack based representation of bytecode
Jimple	Simple, typed, 3-address stackless representation
Shimple	Static Single Assignment (SSA) version of Jimple
Grimp	Jimple with aggregated expressions
Dava	representation used for decompiling dava

Table 2.3: Soot intermediate representations

Statement	Description
NopStmt	The *no operation* statement.
IdentityStmt	Used to bind locals to method parameters and the `this` pointer
AssignStmt	Assignment
IfStmt	Conditional jump
GotoStmt	Unconditional jump
TableSwitchStmt	Table based jump
LookupSwitchStmt	List based jump
InvokeStmt	Method invocation
ReturnStmt	Return statement for non-void methods
ReturnVoidStmt	Return statement for void methods
ThrowStmt	Throws an exception
EnterMonitorStmt, ExitMonitorStmt	Mutual exclusion

Table 2.4: Jimple Statements

Jimple has only 15 kinds of statements, as compared to the over 200 kinds of instructions of the Java bytecode. The main reason is that Java provides specialized versions of its statements for different types, and Jimple avoids this duplication.

The whole weaving process in `abc` is done on the Jimple level, and the stackless representation greatly simplifies the programmatic manipulation of Java bytecode. The use of a higher level representation sets `abc` apart from `ajc`, which directly manipulates stack based code[4].

Table 2.4 lists the available Jimple statements.

Locals Jimple locals are always typed. Locals do not directly correspond to Java slots. When converting Jimple to bytecode, the same slot can be used for multiple locals if the live ranges of these locals do not overlap.

Jimple methods The first statements in the statement chain are generally the identity statements. The first identity statement assigns the **this** parameter to a local. The following identity statements assign the method parameters to designated locals. A method must end with a return statement.

Listing 2.8 shows a simple example of Jimple source code based on the Java class in listing 2.2. The example shows that despite its low level nature, Jimple's syntax resembles that of Java.

Listing 2.8: Jimple example

```
1  public class Foo extends java.lang.Object
2  {
3      public static void main(java.lang.String[])
4      {
5          java.lang.String[] args;
6          java.lang.Object retval;
7
8          // bind parameter 0 to the args local
9          args := @parameter0: java.lang.String[];
10         // invoke the static method bar and
11         // assign the result to a local
12         retval =
13             staticinvoke <Foo: java.lang.Object bar()>();
14         return;
15     }
16     // ...
17 }
```

Chapter 3

Weaving Around-Advice

This chapter describes the code generation strategy of the around weaver. The description focuses on the principles of the code generation and the generated output. The following chapter discusses the implementation in more detail.

3.1 Aims

The aim for `abc` is to implement a generic weaving method that works well for all cases. This is in contrast to `ajc`, which uses one optimized method for certain cases and a generic but inefficient method for other cases.

The rationale behind the chosen approach for `abc` matches the overall project philosophy: the emphasis is on a clean structure and correct but not necessarily most efficient code generation in the first stage. Using the powerful Soot framework, the generated code is optimized further at a later stage. Thus, the emphasis is on generating code that can be optimized later by soot, instead of generating optimized code in the first place.

As [3] shows, certain uses of AspectJ can result in closure object creation. Further experimentation with `ajc`'s generated code made it apparent that object creation imposes a major performance penalty and is to be avoided wherever possible.

The presented solution achieves this goal in all but one special case: Circular advice applications, that is pieces of advice applying to each other's execution or advice applying to its own execution, may require closure creation.

However, these cases of circular advice are usually a program error that results in a non-terminating program. Where this is not the case, the presented method still avoids closure creation in many cases as is described below in detail.

3.2 Concepts

3.2.1 The Joinpoint Shadow

As mentioned, the joinpoint shadow is the range of statements in a program that corresponds to a joinpoint. In `abc`, weaving is done at the *Jimple* level, and the joinpoint shadow is represented as a continuous sequence of Jimple statements.

Before the weaving stage, all shadows are processed to fulfil certain constraints. Particularly, the control flow always enters the shadow at the top and leaves the shadow at the bottom. This means there are no jumps from outside the shadow into the shadow and no jumps from inside the shadow outside the shadow. The shadow does not contain any return statements. This invariant is true for all initial shadows, and it must also be maintained throughout the weaving process. The reason is that multiple pieces of advice can apply to the same shadow, which means that the weaver may encounter shadows which have already been woven into.

Every shadow follows the same format. The beginning and end are marked with a *nop* statement. These *nop* statements are useful for marking positions within a sequence of statements and do not have any side effects. The later optimization steps remove all *nops*.

Figure 3.1 shows a joinpoint shadow at the *Jimple* level. This particular shadow belongs to the call of the method Foo.bar(). The call has simply been surrounded by two *nop* statements.

Listing 3.1: Joinpoint shadow at the Jimple level

```
1  public class Foo extends java.lang.Object
2  {
3      public static void main(java.lang.String[])
4      {
5          java.lang.String[] args;
6          java.lang.Object retval;
7
8          args := @parameter0: java.lang.String[];
9          nop; // beginning of shadow
10         retval =
11             staticinvoke <Foo: java.lang.Object bar()>();
12         nop; // end of shadow
13         return;
14     }
15     // ...
16 }
```

Listing 3.2: Example: Method with multiple returns

```
1 public int fact(int n) {
2     if (n==0)
3         return 1;
4     else
5         return n*fact(n-1);
6 }
```

The listing below shows the execution shadow of the factorial method above.

This shadow requires pre-processing since there are multiple exit points within the shadow.

The listing on the left is the original statement sequence of the shadow. The listing on the right shows the shadow after the pre-processing step. The two return statements have been replaced by a single return statement. Note that this return statement is positioned right after the shadow and not within the shadow itself.

Original shadow statements	Processed shadow statements

`public int fact(int)`
`{`
`  Foo this;`
`  int n, $i0, $i1, $i2;`
 	
`  this := @this: Foo;`
`  n := @parameter0: int;`
 	
`  if n != 0 goto label0;`
 	
`  return 1;`
 	
`  label0:`
`  $i0 = n - 1;`
`  $i1 = virtualinvoke this.`
`  <Foo: int fact(int)>($i0);`
`  $i2 = n * $i1;`
 	
`  return $i2;`
`}`
	`public int fact(int)`
	`{`
	`  Foo this;`
	`  int n, $i0, $i1, $i2;`
	
	`  this := @this: Foo;`
	`  n := @parameter0: int;`
	`  nop; // beginning of shadow`
	`  if n != 0 goto label0;`
	
	`  $i2 = 1;`
	`  goto label1;`
	
	`  label0:`
	`  $i0 = n - 1;`
	`  $i1 = virtualinvoke this.`
	`  <Foo: int fact(int)>($i0);`
	`  $i2 = n * $i1;`
	`  label1:`
	`  nop; // goto target`
	`  nop; // end of shadow`
	`  return $i2;`
	`}`

3.2.2 Advice Methods

The front-end of the compiler creates a Java method for each piece of advice. In the case of around advice, the method has the return type of the advice itself. The body of the Java method consists of the advice body with some modifications.

- **proceed** statements are converted to calls to a placeholder Java method. The first parameters of the method correspond to the parameters of the advice declaration.
- For keywords that retrieve context, such as **thisJoinPoint**, extra parameters are added to the advice method. Occurences of the keywords are then replaced by the corresponding locals.
- Further parameters are added to pass context information. This is described in more detail in a later section.

3.2.3 Advice Applications

The *matcher* stage of the compiler computes a list of *advice applications*. An advice application is essentially a pair of advice method and joinpoint shadow with some additional information.

3.2.4 Dynamic residue

Pointcut expressions pick out joinpoints in the program. Conceptually, at compile time, for each poincut all joinpoint shadows in the program are considered. A static analysis is performed with three possible outcomes: The pointcut could either match *always*, *never* or *sometimes*.

The *sometimes* match case means it cannot be statically determined if the pointcut matches and dynamic tests at runtime are required. These dynamic tests are the *dynamic residue* of the advice application[7].

In the case of multiple variable bindings, the dynamic residue may also contain tests that determine at runtime which context variable to bind to the advice formals.

The dynamic residue is determined at earlier stages before the weaving process. The weaver sees the residue as part of the advice applications. The residue is represented as an abstract syntax tree and can generate the corresponding *Jimple* statements.

3.3 Weaving

3.3.1 Basic weaving strategy

One advice method can apply to many shadows throughout the program. In order to avoid code duplication, the aim is to use this one method for

all advice applications. Furthermore, the method should reside in the class corresponding to its original aspect. This avoids visibility problems that could arise when moving the method into a different class.

When applying *around* advice to a shadow, the first step is to move the shadow out of the original method. Effectively, the shadow is replaced by the following code, shown as simplified pseudo code:

Listing 3.3: Structure of the shadow replacement

```
dynamic residue code
...
if (dynamic residue passed)
    call advice method
else
    call original shadow
```

If the residue check passes, the advice is invoked. If the check fails, the original shadow has to be executed directly.

The main challenge arises out the fact that within the advice method, each **proceed** statement has to invoke the original shadow. Since one advice method can apply to many shadows, the **proceed** statements must have polymorphic behaviour.

Each advice application modifies the joinpoint shadow. It has to be ensured that the shadow fulfils the invariants regarding entry and exit points because later advice applications may apply to the same shadow.

3.3.2 Joinpoint Context

Joinpoints have context. Some of this context is exposed to the programmer. The language defines for each kind of joinpoint what context is exposed to the programmer and how the programmer can access it. Examples of exposed context are the actual arguments of a function execution, or the target object of a method call.

The dynamic residue is responsible for binding exposed context to advice formals.

However, the weaver has to deal with context on a lower level. As mentioned, the strategy is to move the shadow out of its original method. The context the weaver has to deal with can be defined in terms of the shadow and its locals.

By definition, each shadow has a return value (which can be of type **void**). This is part of the invariant shadows have to obey. Apart from this return value, any number of context values can flow into the shadow. In terms of *Jimple*, these context values are locals which are written to outside of the shadow and read from inside the shadow.

The around weaver retrieves the returned value from the advice application. The other context values are calculated dynamically using the algo-

rithm outlined above: The set of all locals that are written to outside of the shadow is intersected with the set of all locals read from inside the shadow.

For later stages of the weaver, it is desirable for the calculated context to include all exposed context values. A problem arises if literals are used in method invocations or field sets. In this case, there is no local that can be discovered by the described dynamic context algorithm. To solve this problem, the matching stage of the compiler was modified to introduce locals for these literals. The following example illustrates the transformation. The shadow is a simple method call with two arguments passed as literals.

Listing 3.4: Joinpoint shadow with literals

```
1 ...
2 nop; // beginning of shadow
3 staticinvoke <Foo: void foo(String, int)>("test", 42);
4 nop; // end of shadow
5 ...
```

The transformation creates two new locals that are assigned the literals. These assignments are inserted before the shadow. This way, the context discovery algorithm will identify these locals as context.

Listing 3.5: Joinpoint shadow with unique locals

```
1 String uniqueLocal1;
2 int uniqueLocal2;
3 ...
4 uniqueLocal1="test";
5 uniqueLocal2=42;
6 nop; // beginning of shadow
7 staticinvoke
8    <Foo: void foo(String, int)>(uniqueLocal1, uniqueLocal2);
9 nop; // end of shadow
10 ...
```

The Soot framework automatically removes unnecessary locals at a later optimization stage.

The necessity of calculating the context values dynamically arises out of the fact that multiple advice applications can apply to one shadow. Each application is free to modify the shadow in any way as long as the shadow invariants are obeyed. This means additional context values could have been introduced by an earlier advice application, and the dynamic calculation finds these context values.

In the case of around advice, the context values generally have to somehow be passed through the advice method to the original shadow. The mechanism of binding advice arguments to context values can be regarded as an interception of this context information that complicates matters further.

3.3.3 The Closure Approach

The obvious solution to achieve the described behaviour of **proceed** is polymorphism using closure objects. We briefly review the details of such an implementation based on closures, as well as the problems it introduces. We then proceed (no pun intended) to solving those problems through an alternative implementation strategy.

For each advice method, an interface type has to be defined. This interface has a proceed method signature matching the advice method.

Listing 3.6: Closure interface

```
1 public interface AroundClosure$1
2 {
3     public [ret-type] proceed$1([arg-type] arg1, ...);
4 }
```

A new parameter for the closure has to be added to the advice method. Every proceed call within the advice method is replaced by a call to the proceed method of the closure.

Listing 3.7: Advice method, closure approach

```
1 [ret-type] adviceMethod$1(AroundClosure$1 closure,
2                           [arg-type] arg1, ...)
3 {
4     ...
5     [ret-type] result=closure.proceed$1(arg1'', ...);
6     ...
7     return result;
8 }
```

For each shadow, a new closure type is created that implements the interface. The proceed method then has to execute the original shadow.

Listing 3.8: Implementation of closure interface

```
1 public class AroundClosure$1$Implementation$1
2                     implements AroundClosure$1
3 {
4     public [ret-type] proceed$1([arg-type] arg1, ...) {
5         ... do what the shadow did...
6     }
7 }
```

The original shadow is replaced by code that creates an instance of a closure object and passes it to the advice method (assuming no dynamic residue).

Listing 3.9: Closure creation at the shadow site

```
1  public class ShadowClass
2  {
3      public void shadowMethod()
4      {
5          nop; // beginning of shadow
6          AroundClosure$1 closure=new
7                  AroundClosure$1$Implementation$1();
8          ... // store additional context information
9          Aspect.aspectOf().adviceMethod$1(closure, arg1, ...);
10         nop; // end of shadow
11     }
12     ...
13 }
```

As mentioned, this approach is rather expensive at runtime due to the object creation.

3.3.4 Avoiding the Closure

This thesis presents two different solutions to avoid this closure creation.

Object Reuse and the Shadow-ID

The first solution is based on the following idea: Instead of creating a new closure object, the existing object associated with the shadow could be reused.

To achieve this, the class in which the shadow resides can implement the closure interface. When calling the advice method, **this** can simply be passed instead of passing a new closure object:

Listing 3.10: Reusing the shadow's instance

```
1  public class ShadowClass implements AroundClosure$1
2  {
3      public [ret-type] proceed$1([arg-type] arg1, ...) {
4          ... // do what the shadow did
5      }
6      public void shadowMethod()  {
7          Aspect.aspectOf().adviceMethod$1(this,
8              arg1, ...);
9      }
10 }
```

A problem has to be addressed: The same advice can apply to multiple shadows in the same class, but the class can only implement the proceed method of the interface once.

The solution is to this problem is the *Shadow-ID*. The Shadow-ID is an integer constant that uniquely identifies each shadow application applying the same advice method to the same class.

The closure interface has to be modified to expect the Shadow-ID as one of the parameters of the **proceed** method.

Also, the advice method has to have a Shadow-ID parameter. The advice method must pass this Shadow-ID to all **proceed** invocations in its body.

At the site of the original shadow, a unique integer constant is passed to the advice method as shown below:

Listing 3.11: The Shadow-ID mechanism

```
1  public class ShadowClass implements AroundClosure$1
2  {
3      public [ret-type] proceed$1(int shadowID,
4                                  [arg-type] arg1, ...) {
5          switch(shadowID) {
6          case 0:
7                ... // do what the first shadow did...
8          case 1:
9                ... // do what the second shadow did...
10         }
11     }
12     public void shadowMethod()
13     {
14         // Unique Shadow-ID of this shadow: 0
15         Aspect.aspectOf().
16                adviceMethod$1(this, 0, arg1, ...);
17     }
18     public void anotherShadowMethod()
19     {
20         // Unique Shadow-ID of this shadow: 1
21         Aspect.aspectOf().
22                adviceMethod$1(this, 1, arg1, ...);
23     }
24 }
```

Another problem that has to be addressed is inheritance: The same advice can apply to two classes where one class directly or indirectly inherits from the other. In this case, the access method of the sub-class overrides the proceed method of the super-class.

To deal with this, all shadows within these classes must have unique Shadow-IDs. The solution then is to modify the proceed methods to call the overridden method in case of an unknown Shadow-ID. This guarantees that eventually the correct proceed method is called.

Listing 3.12: Shadow-ID and inheritance

```
1  public class ShadowClassExt extends ShadowClass
2                       implements AroundClosure$1
3  {
4      public [ret-type] proceed$1(int shadowID,
5                                  [arg-type] arg1, ...) {
6          switch(shadowID) {
7          case 2:
8              ... // do what the shadow did...
9              break;
10         default: // default to super class
11             super(shadowID, arg1, ...);
12         }
13     }
14     public void anotherShadowMethod()
15     {
16         Aspect.aspectOf().
17                 adviceMethod$1(this, 2, arg1, ...);
18     }
19 }
```

The **super** call is a non-virtual `staticinvoke` statement at the bytecode level and thus relatively cheap.

Another problem arises when the joinpoint shadow is in a static method: This proposed solution of object reuse can only be used if there is such an object. This is only the case for shadows residing in a non-static method.

One possible but unsatisfactory solution would be to resort to closure object creation for this case. A possibly better compromise would be to create a singleton closure instance that can be reused.

However, both of these solutions are far from optimal. This led to the creation of another strategy which turns out to be general enough to cover the non-static cases as well.

The Static Class ID

The second strategy does not need any closure interface. Instead, the **proceed** method to which the shadow is moved is static.

A unique integer constant is generated for each class to which a certain piece of advice applies. This is called the *Static Class ID*.

The reader may recall that for the previous method, each occurrence of **proceed** within the advice method was replaced by a call to proceed on the closure object:

```
1 [ret-type] adviceMethod$1(AroundClosure$1 closure,
2                           int shadowID, [arg-type] arg1, ...)
3 {
4     ...
```

```
5     closure.proceed$1(shadowID, arg1, ...);
6     ...
7 }
```

For the *Static Class ID* method, a new parameter is added to the advice method, namely the Static Class ID. Each proceed invocation is replaced by a switch statement that dispatches to the correct proceed method based on the Static Class ID.

```
1 [ret-type] adviceMethod$1(AroundClosure$1 closure,
2                           int shadowID, int staticClassID,
3                           [arg-type] arg1, ...)
4 {
5     ...
6     switch (staticClassID) {
7     case 0: closure.proceed$1(shadowID, arg1, ...);
8             break;
9     case 1: ShadowClass.proceed_s$1(shadowID, arg1, ...);
10            break;
11    case 2: ShadowClass2.proceed_s$1(shadowID, arg1, ...);
12            break;
13    ...
14    }
15    ...
16 }
```

Note that the Static Class ID zero is a special case: To support both described weaving methods simultaneously, an ID of zero dispatches to the closure proceed. For non-zero Static Class IDs, the closure is simply **null**. All other IDs dispatch to the corresponding static proceed method.

At the shadow location, the Static Class ID is passed to the advice method:

```
1 public class ShadowClass2
2 {
3     public static[ret-type] proceed_s$1(int shadowID,
4                     [arg-type] arg1, ...) {
5         switch(shadowID) ... // as before ...
6     }
7     public static shadowMethod()
8     {
9         Aspect.aspectOf().adviceMethod$1(
10            null, // closure is null
11            0,    // shadow ID
12            2,    // Static Class ID assigned to this class
13            arg1, ...);
14    }
15    //...
16 }
```

Source type	Target type
all reference types	java.lang.Object
double	**double**
float	**float**
long	**long**
int	**int**
short	**int**
boolean	**int**
char	**int**

Table 3.1: Context parameters, mapping of types

3.3.5 Passing Context

The previous discussion of avoiding closures has largely ignored passing of context. As described earlier, joinpoints have context that consists of a return value and the locals the shadow depends on. Since the shadow is moved from its original location to the proceed method, the context has somehow be passed from the original site to that method.

To avoid expensive object creations, the aim is to transfer context on the stack. Since Java only allows references to heap locations and not to the stack, the context has to be passed explicitly using method arguments.

Generally, every joinpoint shadow has different context in terms of the number and types of context values. The solution is to create a minimal set of parameters that covers all the shadows to which the advice method applies. In order to reduce the number of necessary parameters, the set of all types is mapped to a small, constant set (see Table 3.1). Particularly, all reference types are mapped to Object and the types **short**, **boolean** and **char** are all mapped to **int** since they are largely treated like **int** types by the JVM[8].

The following example illustrates the technique. The around advice applies to two different method calls. Each of these calls has a different signature, and consequently different context values have to be transferred to the shadow.

Listing 3.13: Context parameters, example program

```
1 public class Foo
2 {
3     public static void main(String args[])
4     {
5         new Foo().bar1("test");
6         new Foo().bar2(1.0d);
7     }
8     public void bar1(String s) {}
```

```
9       public void bar2(double d) {}
10  }
11  aspect Aspect
12  {
13      void around(): call(void *.bar*(..))
14      {
15          proceed();
16      }
17  }
```

Both shadows have a context value of type Foo that is the target of the method call. This value is mapped to the Object type. In addition, the first method call has a context value of type String. This value is also mapped to the Object type. The second method call has a context value of type **double**, which is mapped to itself.

This means that two parameters of type Object and one of type **double** are needed to accommodate both shadows.

The listing below shows the proceed method with the added context arguments. The calls to the advice method have been modified to pass the context values. Note that for both cases, there is one unused context parameter. For these unused parameters, the default value of the respective type is passed.

Listing 3.14: Context parameters in the shadow class

```
1   public class Foo
2   {
3       public static void proceed$1(int shadowID,
4               java.lang.Object contextArg1,
5               double           contextArg2,
6               java.lang.Object contextArg3)
7       {
8           switch (shadowID)
9           {
10              case 0: ((Foo)contextArg1).test2(contextArg2);
11                  return;
12              case 1: ((Foo)contextArg1).test1(contextArg3);
13                  return;
14              default: throw new RuntimeException();
15          }
16      }
17      public static void main(java.lang.String[] r0)
18      {
19          Foo target1 = new Foo();
20          Aspect.aspectOf()
21              .adviceMethod$1(1, 1, target1, 0.0, "test");
22          Foo target2 = new Foo();
23          Aspect.aspectOf()
24              .adviceMethod$1(0, 1, target2, 1.0, null);
```

```
25          return;
26      }
27      ...
28  }
```

In the advice method, the context parameters are passed to all the proceed calls:

Listing 3.15: Context parameters in the advice method

```
1  class Aspect
2  {
3      final void adviceMethod$1(int shadowID,
4              java.lang.Object contextArg1,
5              double           contextArg2,
6              java.lang.Object contextArg3)
7      {
8          ...
9          Foo.proceed$1(shadowID
10                 contextArg1,
11                 contextArg2,
12                 contextArg3
13                 );
14         ...
15         return;
16     }
17     ...
18 }
```

3.3.6 Binding of Exposed Context Values

As described earlier, the programmer can access exposed values of the joinpoint context. This is done by declaring advice formals and binding the context values to the advice formals. The bindings are defined in the pointcut expression using **this**, **target** and **args**.

The dynamic residue that is passed to the weaver generates the binding code that assigns the correct context values to the advice formals.

The proceed method has formal parameters corresponding to the advice formals. Depending on the binding, these formals must be assigned to the correct context values of the shadow.

The weaver analyzes the dynamic residue to determine the possible variable bindings. For most cases, these bindings can be determined statically. This means that if the residue succeeds, there is a fixed mapping between advice formals and context values. For these cases, the proceed method simply contains the necessary assignments to assign the advice formals to the context values.

A problem arises if the residue fails. In this case, there are no bindings and the expected behaviour is that the shadow is executed with its original

context. One way to achieve this would be to duplicate the shadow and simply execute the original shadow in the failed case. However, one of the design goals of the weaver is to avoid code duplication. This lead to the introduction of the *skip flag*.

The Skip Flag

As explained, the joinpoint shadow is moved into the proceed method. If the dynamic residue of the joinpoint succeeds, the advice method is invoked which in turn calls the proceed method. If the residue fails, the shadow has to be executed directly without invoking the advice method.

The *skip flag* enables the proceed method to handle both the pass and fail cases of the dynamic residue. The flag indicates to the method whether it was invoked by the advice method, in which case the advice formals are bound to context values, or if it was invoked directly, in which case the advice formals are not bound and the standard context parameters should be used.

The following example is intended to illustrate this mechanism. Consider the following simple base program and aspect.

```
1  public class Foo
2  {
3      public static void main(String args[])
4      {
5          new Foo().bar(0);
6      }
7      public void bar(int i) {}
8  }
```

```
1  aspect Aspect
2  {
3      void around(int intArg):
4          call(void *.bar*(..)) &&
5          args(intArg) &&
6          target(Foo)
7      {
8          proceed(intArg);
9      }
10 }
```

For the around advice, the argument of the call is bound to the advice formal intArg. In the example, the **target** primitive is not used to bind an advice formal: When given a type, **target** matches dynamic joinpoints with a target object of the given type.

The code below shows the implementation of the *skip flag*. Based on the flag, the proceed method either assigns the context parameter or the advice

formal. The context value representing the target object is never bound to any advice formal and so is always assigned the context parameter.

Listing 3.16: Skip flag: proceed method

```
1  public class Foo
2  {
3      public static void proceed$0(
4          int intArg, // advice formal
5          int shadowID,
6          boolean skipFlag,
7          java.lang.Object contextArg1,
8          int contextArg2 ){
9
10         switch(shadowID) {
11         case 0:
12             int arg;
13             if (skipFlag)
14                 arg=contextArg2; // unbound case
15             else
16                 arg=intArg;       // bound case
17
18             Foo callTarget=(Foo)contextArg1; // never bound
19             callTarget.bar(arg);
20             break;
21         default:
22             throw new RuntimeException();
23         }
24     }
25     ...
26 }
```

At the shadow site, the dynamic residue performs a type check and, based on the result, either invokes the proceed method directly with the skip flag set to **true** or invokes the advice method. [1]

Listing 3.17: Skip flag: shadow site

```
1  public class Foo {
2      ...
3
4      public static void main(String args[])
5      {
6          Foo foo=new Foo();
7          int i=0;
8          if (foo instanceof Foo) {
9              // residue passed
10             Aspect.aspectOf().adviceMethod$0(
```

[1] For this simple case, the optimizer of abc would actually be able to determine the outcome of the type check at compile time.

```
11                  i,      // advice formal
12                  null,   // closure
13                  0,      // shadow ID
14                  1,      // static class ID
15                  foo,    // context argumenttrue
16                  0       // context argument (default)
17          } else {
18              // residue failed
19              proceed$0(
20                  0,       // advice formal (default)
21                  0,      // shadow ID,
22                  true,   // skip flag
23                  foo,    // context argument
24                  i       // context argument
25                  )
26          }
27      }
28      public void bar(int i) {}
29  }
```

The advice method in turn calls the proceed method with the skip flag set to **false**.

Listing 3.18: Skip flag: advice method

```
1  aspect Aspect
2  {
3      void adviceMethod$0(
4              int intArg,
5              Around$closure$0 closure,
6              int shadowID,
7              int staticClassID,
8              Object contextArg1,
9              int  contextArg2) {
10               Foo.proceed$0(
11                   intArg,
12                   shadowID,
13                   false, // skipFlag
14                   contextArg1,
15                   contextArg2);
16      }
17 }
```

The bind mask

The described method of binding advice formals assumed that the mapping between advice formals and context values can be determined at compile time. This is generally the case if for each advice formal there is exactly one binding primitive in the pointcut. However, the pointcut language does not

make this restriction and it is easy to create expressions that do not fit this pattern.

The following are simple examples.

```
1 void around(Foo x) : this(x) || target(x)
2 void around(Foo x) : args(x,..) || args(..,x)
```

In many of these cases, the actual binding is only known at runtime. The dynamic residue takes care of assigning the correct context values to the advice formals.

However, as mentioned, in the case of *around* advice there is a complication. In the proceed method, the advice formals have to be assigned back to the right context values. If the bindings are only known at runtime, these assignments also have to be determined at runtime.

The proposed solution to this problem is the *bind mask*. [2] The aim was to find an efficient method to transfer the actual binding information at runtime from the shadow site to the proceed method. The bind mask is a bitmask that has this information encoded. This bitmap is passed from the shadow to the advice method, which passes it on to the proceed method.

The bind mask only contains information for parameters for which the binding cannot be determined statically at compile time. To reduce the overall number of parameters, the *skip flag* described in the last section is encoded in the bind mask.

In order to create the bind mask, a transformation of the residue AST is performed. For each ambiguous *Bind* node, a *BindMaskResidue* node is inserted. The *BindMaskResidue* sets the appropriate bits in the bind mask and then generates the original *Bind* residue.

The implementation ignores structural information of the pointcut expressions. A deeper analysis of the expression combined with a different bind mask representation would in some cases result in a more efficient use of the bind mask. As an example, consider the following pointcut:

```
1 pointcut(Foo x, Object y):
2     (args(x,..) && this(y)) || (args(..,x) && target(y))
```

The outcome of the bindings depends on which operand of the || operator succeeds. This information can be encoded in one bit. However, this implementation chooses not to use such structural knowledge and instead encodes the binding information for each advice formal separately, resulting in the use of two mask bits. This was a design choice: Extensibility is a key feature of `abc`. Using structural information for the bind mask would make it more difficult to modify and extend the pointcut language because

[2] As mentioned, the official AspectJ compiler ajc does not support this behaviour and aborts with a compiler limitation error if presented with these kinds of pointcut expressions.

for every added construct, the validity of the structural analysis would have to be ensured.

Creation of the bind mask For each advice formal, there are $n \geq 1$ different bindings. Information is only encoded for advice formals with $n > 1$.

For each parameter, a number of dedicated bits is reserved in the bind mask.

This is done for efficiency reasons. This way, the binding information for a given parameter can be extracted with simple *shift* and *binary and* operations, and no division is required.

The ambiguity limits of the 31 bits used for encoding can be illustrated by the following cases:

- One variable could be ambiguously bound to 2^{31} context values.
- 32 variables could each be ambiguously bound to 2 context values.

As an example, consider the following aspect. Depending on the type, the pointcut binds either the first or the last argument of the method call to the advice formal s. The advice then passes a changed value to **proceed**.

Listing 3.19: Multiple bindings, proceed

```
1 aspect Aspect
2 {
3     void around(String s):
4         call(void *.foo*(..)) &&
5         (args(s,..) || args(..,s))
6     {
7         proceed("new");
8     }
9 }
```

The following base program contains two method calls that trigger the advice with a different variable binding.

Listing 3.20: Base program triggers different bindings

```
1 public class Foo
2 {
3     public static void main(String args[])
4     {
5         new Foo().foo("string", null);
6         new Foo().foo(null, "string");
7     }
8     public void foo(Object ob1, Object ob2) {
9         System.out.println(ob1 + ", " + ob2);
10    }
11 }
```

The output of the compiled program shows that for each call, the correct context value has been replaced:

```
new, null
null, new
```

To achieve this, the bind mask is created at the shadow site. In this example, bit zero of the bind mask is used as the skip flag and bit one indicates whether the first or the last argument is bound.

Listing 3.21: Creation of the bind mask

```
1  public class Foo
2  {
3     public static void main(String args[])
4      {
5        Foo foo=new Foo();
6        Object arg1=''string'';
7        Object arg2=null;
8        nop; // beginning of first shadow
9        String adviceFormal;
10       int bindMask=0; // initialization
11       label_0:
12       {
13         if (arg1 instanceof String) {
14           adviceFormal=arg1;
15           bindMask|=0; // removed by optimizer
16         } else {
17           if (arg2 instanceof String) {
18             adviceFormal=arg2;
19             bindMask|=2; // set bit 1
20           } else { // skipped case
21             bindMask=1; // set skip flag
22             adviceFormal=null;
23             proceed_s$0(adviceFormal,
24                         0,      // shadow ID
25                         bindMask,
26                         foo,    // contextArg1
27                         arg1,   // contextArg2
28                         arg2); // contextArg3
29             break label_0;
30           }
31         }
32         Aspect.aspectOf().adviceMethod$0(
33                                          adviceFormal,
34                                          null, // closure
35                                          0,    // shadow ID
36                                          1,    // static class ID
37                                          bindMask,
38                                          foo,  // contextArg1
```

```
39                                          arg1, // contextArg2
40                                          arg2);// contextArg3
41        }
42        nop; // end of first shadow
43        ...
44        // second shadow...
45      }
46      ...
47  }
```

The proceed method performs the assignments based on the bind mask.

Listing 3.22: Evaluation of the bind mask

```
1  public class Foo
2  {
3    ...
4    public static void proceed_s$0(String s,
5                                    int shadowID,
6                                    int bindMask,
7                                    Object contextArg1,
8                                    Object  contextArg2,
9                                    Object contextArg3) {
10     switch(shadowID) {
11       case 0:
12         Object arg1;
13         Object arg2;
14         if (bindMask==1) { // skip case
15           arg1=contextArg2;
16           arg2=contextArg3;
17         } else {
18           // first assign the default context
19           arg1=contextArg2;
20           arg2=contextArg3;
21           // then overwrite the bound value
22           // with the advice formal.
23           switch ((bindMask & 2) >> 1)
24           {
25             case 0:
26               arg1 = s;
27               break;
28             case 1:
29               arg2 = s;
30               break;
31             default:
32               throw new RuntimeException();
33           }
34         }
35         Foo foo(Foo)contextArg1; // never bound
36         foo.foo(arg1, arg2);
37       case 1:
```

```
38            ...
39        }
40
41    }
42 }
```

One noteworthy detail is that when evaluating the bind mask, the weaver first assigns all the default context values and then overwrites the bound value based on the bind mask.

An alternative strategy would be to assign all values within the switch statement:

Listing 3.23: Alternative evaluation strategy

```
1  switch ((bindMask & 2) >> 1)
2  {
3    case 0:
4      arg1 = s;
5      arg2 = contextArg3;
6      break;
7    case 1:
8      arg2 = s;
9      arg1 = contextArg2;
10     break;
11   ...
12 }
```

However, this leads to a quadratic number of assignment statements in terms of the number of context values involved. If n context values are bound to one advice formal, the above switch statement would contain n cases with n assignments each, resulting in n^2 overall assignment statements. With the strategy used by the weaver, there would be n default assignments plus n cases with one assignment each, resulting in $2n$ overall assignment statements. At runtime, the presented strategy results in the execution of one additional assignment which is negligible and clearly preferable over quadratic code size.

3.3.7 Local and anonymous classes

Local and anonymous classes can be declared within the advice method. Since these classes are in the scope of the advice method, they can contain **proceed** statements. This offers a way to delay the execution of the proceed statement to a point after the advice method has returned: An instance of the local or anonymous class can be stored elsewhere and the method containing the **proceed** statement can be invoked later (see the earlier thread safety example).

This means that the joinpoint context has to be kept. Experiments show that the official AspectJ compiler `ajc` resorts to closure creation in this case.

However, with the weaving method described so far, it is possible to use the instance of the local or anonymous class itself as the closure object and thus avoid closure object creation. To accomplish this, the weaver adds data members to every local or anonymous class. These contain the joinpoint context, the bind mask, the shadow id, the closure reference and the static class ID.

The following is a simple example of a local class within *around* advice.

Listing 3.24: Local class in around advice

```
1  aspect Aspect {
2    void around(final Object ob) :
3         call(void *.*(..)) &&
4         this(ob)
5    {
6      class FirstDegree implements Runnable {
7        void run() { proceed(ob); }
8      }
9      Runnable r=new FirstDegree();
10     r.run();
11   }
12 }
```

Since the Java Virtual Machine (JVM) does not natively support any type of nested classes, the compiler translates the local class into a normal Java class. The weaver adds members for all the required values to the class and, using these values, replaces each **proceed** statement with the usual construct as presented earlier.

Listing 3.25: Modified local class

```
1  class Aspect$FirstDegree implements java.lang.Runnable {
2     private final Object val$ob;
3     private final Aspect this$0; // reference to
4                                  // containing aspect
5     public int shadowID;
6     public int staticClassID;
7     public Around$Closure$0 closure;
8     public int bindMask;
9     public Object contextArg1;
10
11    public Aspect$FirstDegree(Aspect aspect, Object ob) {
12      this.this$0=aspect;
13      this.val$ob=ob;
14    }
15    public void run() {
16      // proceed replaced by switch statement as before
17      switch(staticClassID) {
18        case 0:
19          closure.proceed$0(shadowID, bindMask, contextArg1);
```

```
20          case 1:
21            Foo.proceed_s$0(shadowID, bindMask, contextArg1);
22          default:
23            throw new RuntimeException();
24        }
25      }
26  }
```

In the advice method, code is inserted after the creation of the local class to initialize the added members.

Listing 3.26: Initialization of local class

```
1  class Aspect {
2    void around(Object ob,
3                Around$Closure$0 closure,
4                int shadowID,
5                int bindMask,
6                Object contextArg1)
7    {
8      Aspect$FirstDegree r2=new Aspect$FirstDegree(this, ob);
9      r2.bindMask=bindMask;
10     r2.shadowID=shadowID;
11     r2.closure=closure;
12     r2.contextArg1=contextArg1;
13     Runnable r=r2;
14     r.run();
15   }
16 }
```

Nesting A further complication which the example above did not illustrate is that local and anonymous classes can be nested. This means a local or anonymous class can contain further local or anonymous classes and so on, and all of these classes can contain **proceed** statements.

For all classes containing **proceed** statements, the context has to be kept. As was illustrated above, this can be done using additional member variables to store the context.

However, it is useful to make a distinction between the outermost classes and the nested classes. In the following discussion, the outermost classes inside the *around* advice body shall be referred to as first degree classes.

A non-static nested class in Java has access to the instance of the enclosing class in which it was instantiated. The implementation achieves this by adding an implicit member variable `this$0` to the nested class that points to an instance of the containing class. This reference guarantees that the instance of the outer class exists for the lifetime of the contained object.

In the context of *around* advice, this means that the instances of first degree classes exist for the lifetimes of all instances of nested classes. As a

consequence, only the first degree classes need to have extra data members to store the context and all nested classes can simply refer to the first degree classes.

Since nesting can occur with an arbitrary depth, a chain of references may have to be followed to reach the first degree object. However, this is only done once within any method, even if multiple members of the first degree object are accessed.

The implementation aims to treat the advice method and all methods contained in local and anonymous classes within that method in a similar manner. Particularly, every method contains locals that correspond to the context values. Depending on the type of method, these locals are assigned different values.

- For the advice method itself, the locals are assigned the corresponding context parameters.
- For first degree methods, the locals are assigned the values of the context member fields of that class.
- For methods in nested classes, the locals are assigned the values of the fields of the enclosing first degree class.

This invariant of having locals corresponding to the context values in every method helps with the issue of initializing the context member fields of the first degree classes.

The instantiation of local and anonymous classes does not necessarily follow the nesting hierarchy. For example, one first degree class could instantiate another first degree class. With the context locals in place in every method, the initialization is a simple matter of searching all methods for instantiation statements. If the instantiation is a first degree class, statements are added directly after the instantiation to assign the context locals of the method to the fields of the newly instantiated object.

3.3.8 Circular advice applications

In the AspectJ language, the execution of advice itself is a joinpoint. This means that advice can apply to the execution of other advice or even to its own execution. In pointcuts, these joinpoints are captured with the **adviceexecution** primitive.

Advice applying to advice can be expressed as a directed graph structure. If this graph is acyclic, the described methods of weaving are sound. However, under certain circumstances, cycles occur in the graph. Advice applying to itself is such a case.

Usually, this is the result of a program error and in that case it usually results in a non-terminating program. However, there are cases where side

effects in the pointcut result in a terminating program despite the cycles in the application graph.

Consider the following example of *around* advice weaving into itself. This program terminates because the expression inside the **if** pointcut primitive has the side effect of decrementing the field n.

Listing 3.27: Program with circular around advice application

```
1  public class Foo
2  {
3      public static void main(String args[])
4      {
5      }
6  }
7  aspect Aspect
8  {
9      public static int n=3;
10     void around(): execution(* Foo.* (..)) ||
11             (adviceexecution() && if(--n>0))
12     {
13         System.out.println("n=" + n);
14         proceed();
15     }
16 }
```

The output of the program is

```
n=0
n=0
n=0
```

The output shows that all the residue checks are performed before any advice is executed. The advice is invoked *around* itself until the residue fails, at which point the advice bodies are executed in a nested fashion.

To further illustrate this behaviour, the output of the aspect was extended.

Listing 3.28: Aspect with circular advice, extended output

```
1  aspect Aspect
2  {
3      public static int n=3;
4      void around(): ... // as before
5      {
6          nesting++;
7          System.out.println(nesting + " =>: n=" + n);
8          proceed();
9          System.out.println(nesting + " <=: n=" + n);
10         nesting--;
11     }
```

```
12      public static int nesting=0;
13  }
```

This aspect produces a more elaborate output (indented for clarity).

```
1 =>: n=0
  2 =>: n=0
    3 =>: n=0
    3 <=: n=0
  2 <=: n=0
1 <=: n=0
```

Such cycles prevent the use of the described weaving method using context parameters. In this case, closure objects are used to close the cycle.

Necessity of closure objects in the presence of cycles Cycles require the creation of closures or a closure like construct. It is impossible to achieve the desired functionality using counters or further context parameters.

One can reason that advice applications of around-advice applying to around advice can form arbitrarily complex directed graphs. The dynamic residue that determines the actual execution of around-advice surrounding around-advice can also be arbitrarily complex, which means no static analysis can generally determine the order of around-advice executions. In a chain of around-advice intercepting around-advice, the semantics of the language dictates that all the dynamic residue is executed before any around advice: The call chain is determined by the dynamic residue, and this chain is then executed *after* all the dynamic residue checks finish. This means that the chain of advice invocations has to be built up and stored until the residue is finished, and such a chain of arbitrary length has to be represented by a stack like structure. Closures offer a clean solution.

3.3.9 Closures

The past sections showed how closure object creation can be avoided in many cases. The first method described showed how the shadow object can be reused as a closure object. For this method, the weaver generates a closure interface and adds a closure parameter to the advice method. This infrastructure can be conveniently used for actual closure objects. The closure class simply implements the closure interface of the advice method, and the closure instance is passed to the advice method using the existing parameter.

The original shadow is moved to a static method within the joinpoint class. This is done for two reasons. First, the strategy is to always keep code in its original class to avoid visibility problems. Second, this is necessary to efficiently deal with the skipped case: If the residue check fails, the original

shadow has to be invoked directly. In this case, the closure creation should be avoided. The static method achieves this and allows the invocation of the original shadow directly without the closure.

The closure's *run* method simply calls the static method in the joinpoint class, passing all the context values as parameters.

For each context value, a dedicated member field of that type is generated in the closure class. This is a major optimization when compared to `ajc`'s implementation. `ajc` creates an array of type Object to store all the context values. This is inefficient for multiple reasons. The array is another object that needs to be created apart from the closure itself. Simple types need to be boxed and unboxed to be stored in the object array. This means another object creation for each value with a simple type. A possibly minor point is that in `ajc`'s case, reference types need to be cast back to their original types using a checked cast.

The closure's fields are public and non-final. Initialization is not done in the constructor to avoid another unnecessary method call. This violation of encapsulation is justified because the closures are completely transparent to the user, thus making efficiency the main concern.

In the example from listing 3.27, the weaver first weaves into the execution of Foo.main() using the normal strategy. For the advice execution part, the weaver uses the closure approach.

The following listing shows the closure class. The proceed method simply calls the static proceed method within the shadow class.

Listing 3.29: Closure class

```
1  public class AroundClosureImpl$0 implements AroundClosure$0
2  {
3      public AroundClosure$0 context0;
4      public int context1;
5      public int context2;
6      public int context3;
7
8      public void proceed$0(int arg1, int arg2)
9      {
10         Aspect.proceed_c$0(
11             arg1,
12             arg2,
13             context0,
14             context1,
15             context2,
16             context3);
17         return;
18     }
19 }
```

Since the shadow of the advice execution joinpoint is in the aspect, the

proceed method is created in the class corresponding to the aspect. The proceed method contains the former body of the advice method.

Listing 3.30: Closure static proceed method

```
1  public class Aspect
2  {
3      public static int n;
4
5      public static void proceed_c$0(
6                              int shadowID,
7                              boolean bindMask,
8                              AroundClosure$0 contextArg1,
9                              int contextArg2,
10                             int contextArg3,
11                             int contextArg4)
12     {
13         System.out.println(n);
14
15         switch (contextArg2)
16         {
17             case 0:
18                 r0.proceed$0(contextArg4, contextArg3);
19                 break;
20             case 1:
21                 Foo.proceed_s$0(contextArg4, contextArg3);
22                 break;
23             default:
24                 throw new RuntimeException();
25         }
26     }
27     ...
28 }
```

The **if** pointcut primitive is converted to a method inside the aspect.

Listing 3.31: Helper function for *if* residue

```
1  public class Aspect
2  {
3    ...
4      public static boolean if$2()
5      {
6          return --n>0;
7      }
8      ...
9  }
```

As shown, the body of the advice method is moved into the proceed method. It is replaced by the code below. If the dynamic residue succeeds, a new closure object is created and initialized. Since in this example the

advice applies to itself, the closure object is passed to a recursive call of the advice method itself. If the residue fails, the proceed method is invoked directly, thus executing the original body of the advice method.

Listing 3.32: Closure object creation

```
1  public class Aspect
2  {
3    ...
4       public final void adviceMethod$0(
5                         AroundClosure$0 closure,
6                         int shadowID,
7                         int staticClassID,
8                         int bindMask)
9      {
10         if (Aspect.if$2()) {
11             Aspect a = Aspect.aspectOf();
12             AroundClosureImpl$0 closureObject =
13                 new AroundClosureImpl$0();
14             closureObject.context3 = shadowID;
15             closureObject.context2 = bindMask;
16             closureObject.context1 = staticClassID;
17             closureObject.context0 = closure;
18             a.adviceMethod$0( // recursive call,
19                 closureObject,// passing the closure object
20                 -1,
21                 0,
22                 0);
23
24         } else { // residue failed: invoke shadow directly
25             Aspect.proceed_c$0(
26                                 -1, // shadow ID, ignored
27                                 1,  // static class ID
28                                 closure,
29                                 staticClassID,
30                                 bindMask,
31                                 shadowID);
32         }
33     }
34 }
```

Chapter 4

Implementation

This chapter gives an overview of the implementation of the weaver at the source code level. The `abc` project and all of its components, namely the Soot framework, Polyglot, JFlex and the CUP parser are written in Java. At the time of writing, `abc` itself consists of about 40.000 lines of code, and the around-weaver code consists of more than 3000 lines of Java. The full source code can be found at the `abc` website[1].

The development of the weaver was carried out in parallel with the other parts of the `abc` project. The overall development of the compiler took about nine months with a team of 13 programmers.

4.1 Structure

Weaving is implemented in an incremental fashion, which means the weavers are invoked separately for each advice application. The around weaver keeps internal data structures to carry over state from one application to the next.

4.1.1 Advice lists

Earlier stages of the compiler compute the advice lists. These are lists of advice applications, which represent the application of a piece of advice to a certain joinpoint shadow and the dynamic residue for that application.

At the weaving stage, the weavers are called for each application on the lists.

4.1.2 Ordering of advice applications

For the around-weaver, the advice applications have to be ordered before they are passed to the weaver. This is necessary to deal with around advice applying to the execution of around advice. As discussed earlier, these applications can be regarded as a directed graph which could contain cycles.

If such cycles exist, the weaver has to use the closure strategy to close the cycles.

As the first step, all these applications that are around-advice applying to around-advice are extracted from the list and their weaving is delayed to the very end. This allows the construction of the graph structure. Then, a topological sort is them performed on the graph. If the sort succeeds, the graph is acyclic and can be woven in the sorted order without resorting to closure object creation.

If the sort fails, the graph contains cycles and a compiler warning is issued. Weaving then continues using the unsorted graph.

Dealing with cycles Whenever the weaver weaves into an around advice method, it sets a flag for that method. Once an advice method has been woven into, it can only be woven using the closure strategy.

4.1.3 Data structures

Table 4.1 lists the main classes within the weaver. Most of these classes wrap Soot entities at the Jimple level such as classes, methods and statements.

Class	Description
AroundWeaver	Top level class.
State	Contains the weaver state.
AdviceMethod	Represents an advice method.
AdviceLocalClass	Represents classes containing code of the advice method. These are nested classes within the advice body and the aspect class itself.
AdviceLocalMethod	A method containing code of the advice method. These are methods of nested classes and the advice method ifself.
NestedInitCall	An initialization statement within an AdviceLocalMethod that initializes a nested class.
ProceedInvocation	An invoke statement representing a **proceed** statement.
ProceedMethod	A proceed method. These methods contain the code of the shadows of advised joinpoints.
AdviceApplicationInfo	Wraps the information associated with an advice application.

Table 4.1: Classes within the weaver

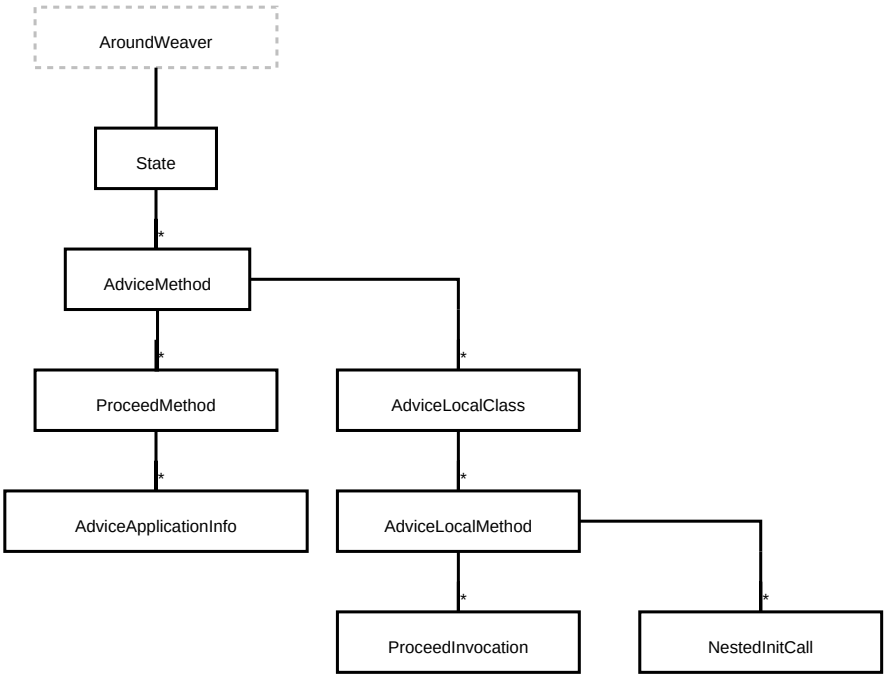


Figure 4.1: Object hierarchy

Figure 4.1 shows the object hierarchy that is formed at runtime.

This hierarchy can be conveniently implemented using Java inner classes. This way, the dynamic object hierarchy is documented in the source code and the inner classes can transparently access members of the enclosing class instance.

The inner class mechanism helped to naturally capture the one-to-many relationships between the weaver entities. For example, advice method objects are accessed at many places throughout the weaver. Using inner classes and having AdviceMethod as the outer-most class, all methods of all inner classes can access the members of the enclosing AdviceMethod instance implicitly.

4.1.4 Incremental weaving

As described, the weaver is invoked once for each advice application. This means that the weaver may have to modify code that was generated for a previous advice application. Particularly, if one advice method is applied to multiple joinpoint shadows, the closure interface associated with the advice method can change with each application because additional context parameters may have to be added. These changes do not only apply to the closure interface but to a number of entities:

- calls to closure interfaces

- proceed methods
- calls to proceed methods
- fields of local and anonymous classes
- initializations of local and anonymous classes.

The weaver has to keep track of these entities between invocations. A complicating factor is that some of these entities may be moved by the weaver in the process of moving a joinpoint shadow to the proceed method. This is done by copying all the statements of the shadow to the proceed method and then deleting the original statements. When doing so, references to the entities mentioned above have to be updated to point to the new instances.

4.2 Changes to the residue classes

4.2.1 Bindings

The residue AST contains Bind nodes that represent a possible binding between an advice formal and a context value. The AST has to be analysed to determine the possible bindings. This is done in two passes. The first pass traverses the residue to determine how many times each advice formal is bound. If any advice formal is bound more than once, the effective binding has to be stored in the bind mask.

The layout of the bind mask is calculated after the first pass. The layout determines the mapping between advice formals and the ranges of bits within the mask representing the bindings of these formals.

The second pass traverses the residue and uses this layout information to insert new residue nodes of type BindMaskResidue at the appropriate places. These nodes generate code that manipulates the bind mask to set the correct bits.

4.2.2 Type check optimization

An optimization was added to the TypeCheck residue. Based on the Java specification [5], an analysis is performed to determine whether a type could possibly be converted to the target type at runtime. If this is not the case, the residue would never succeed and consequently, the advice application does not need to be woven. This optimization further reduces dynamic checks at runtime.

Chapter 5

Benchmarks

This chapter presents some benchmarks of programs produced by `abc` and provides a comparison with `ajc` produced programs. `abc` is now in its final stage of development, and a pre-release version is about to be published. The pre-release status means that the results presented in this chapter are to be treated as somewhat preliminary.

5.1 Methodology

The benchmarks have been carried out on the most recent development version of `abc`. Each test program was compiled with both compilers and then timed. Timing was done with the shell's built in `time` command.

In order to produce sound results, each measurement was carried out ten times. The worst and the best times were ignored in an attempt to remove distortions from other factors such as caching and system activities. The reported result is the average of the remaining times. `ajc` was used in version 1.2.

5.2 Measurements

5.2.1 Coding Standards

A comprehensive study of the performance of AspectJ can be found in [3]. The paper contains five different benchmarks. One of them is based on the *Nullcheck* aspect presented earlier. We use the same benchmark as a basis for our measurements.

In the benchmark, the aspect is applied to a simulator base program. We used a number of different versions of the aspect with changed pointcuts, equivalent to those presented in [3].

Listing 5.1: Nullcheck: Pointcut A

```
1 pointcut methodsThatReturnObjects():
2         call(* *.*(..)) &&
3         !call(void *.*(..));
```

Pointcut *A* matches all calls to non-void methods, including methods returning simple types. Method calls within the aspect itself are also matched. This causes `ajc` to use closures, and both `ajc` and `abc` have to perform boxing and unboxing for simple types.

Listing 5.2: Nullcheck: Pointcut B

```
1 pointcut methodsThatReturnObjects():
2         call(Object+ *.*(..));
```

Pointcut *B* only matches methods that return object types. Again, methods within the aspect are matched and `ajc` uses closures, but no boxing and unboxing is necessary.

Listing 5.3: Nullcheck: Pointcut C

```
1 pointcut methodsThatReturnObjects():
2         call(Object+ *.*(..)) &&
3         !within(lib.aspects..*);
```

Pointcut *C* excludes joinpoints within the aspect itself. This allows `ajc` to use its inlining strategy.

In addition, the benchmark was carried out with a pointcut that does not match any joinpoints, thus effectively measuring the execution time of the base program.

	`ajc`			`abc`		
	default	Soot opt.	forced closures	static (default)	non-static	forced closures
Pointcut A	17.19	17.11	17.10	5.97	6.20	8.53
Pointcut B	5.55	5.59	5.51	1.80	1.97	3.15
Pointcut C	1.50	1.51	5.59	1.81	1.86	3.12
Base program	1.12	1.10	1.11	1.11	1.14	1.12

Table 5.1: Nullcheck execution times (in seconds)

In order to show that the performance gains are not attributable to the Soot optimization framework, `ajc`'s output was run through the Soot optimizer in a separate measurement. The results (Table 5.1, columns 1 and 2) show that the Soot optimizations do not have a significant impact on the outcome of the measurements.

When `abc` and `ajc` are used with their default settings (see Figure 5.1), the biggest improvements can be seen for pointcuts *A* and *B*. For both of

these pointcuts, `ajc` uses its closure based strategy, and `abc` uses the non-closure approach. With pointcut *C*, `ajc` uses its inlining strategy which is slightly faster than `abc`'s generic non-inlining approach.

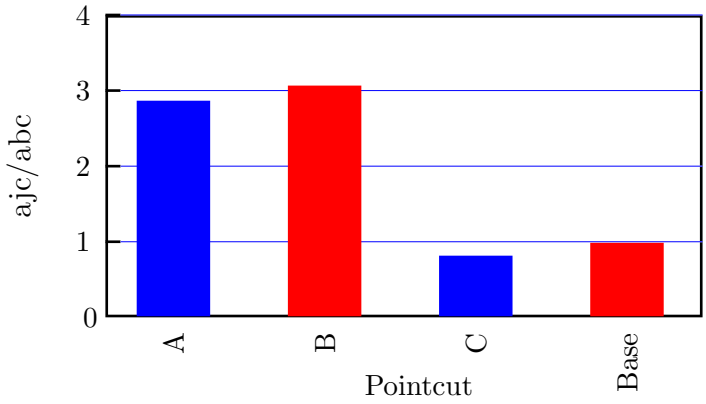


Figure 5.1: abc and ajc, normal operation

To compare the efficiency of the closure implementations, the around weaver was modified to always use the closure strategy and `ajc` was invoked with the `-XnoInline` parameter to force it to use closures. The results (Figure 5.2) show that the optimizations of the closure implementation were worthwhile.

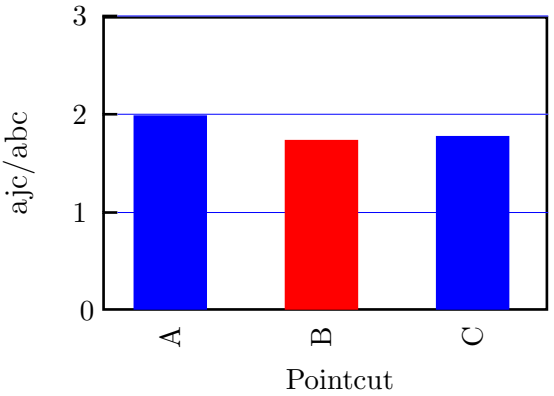


Figure 5.2: ajc and abc forced to use closures

The measurements show that the strategy of always using static proceed methods results in slightly faster programs (Table 5.1, columns 4 and 5). Static proceed methods are therefore used as the default strategy (however, the difference is very small).

5.2.2 Local classes

To measure the performance of the implementation of **proceed** statements in local or anonymous classes, the following simple aspect was created. When applied to a Java program, the aspect captures the program's output during the execution and delays the output till after the program's execution. [1]

Listing 5.4: Benchmark for anonymous classes

```
1  aspect DelayOutput {
2    pointcut outputMethods() :
3      call (* java.io.PrintStream.println (..)) ||
4      ... // more relevant output methods ...
5
6    List outputQueue=new LinkedList();
7
8    void around(): outputMethods()
9    {
10     Runnable worker = new Runnable() {
11       public void run() {
12         proceed();
13       }
14     };
15     outputQueue.add(worker);
16   }
17   after(): execution(public static void main(String[]))
18   {
19     for (Iterator it=outputQueue.iterator(); it.hasNext();){
20       Runnable worker=(Runnable)it.next();
21       worker.run();
22     }
23   }
24 }
```

The aspect was applied to a simple base program that prints a hundred thousand integers in a loop. The measurements are shown in Table 5.2.

In order to remove the overhead of the system output routines, a second measurement was taken that uses an empty placeholder method instead of the system output routines.

	ajc	abc
A: System output	1.75	1.50
B: Output stub	0.69	0.40

Table 5.2: *DelayOutput* results (in seconds)

[1] The listing shows a simplified version of the aspect. A full implementation needs to provide a specialized piece of advice for the `println(java.lang.Object)` overloaded method that calls the object's `toString()` method at invocation time.

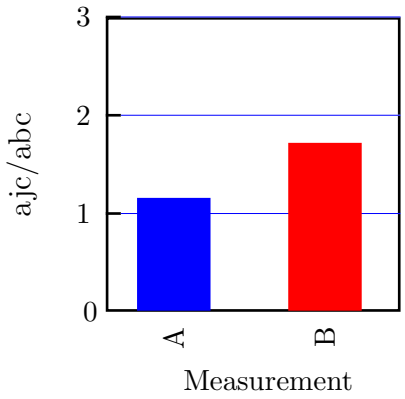


Figure 5.3: *DelayOutput* results, performance ratios

The second measurement shows that the output routines largely dominate the execution time of the first measurement. With this distortion removed, the `abc` implementation is significantly faster.

Chapter 6

Conclusion

6.1 Summary

We presented a compilation strategy for weaving around advice for the AspectJ language. As part of this work, we did a complete implementation of this strategy as part of the AspectBench Compiler `abc`.

The aim of the `abc` project is to create a production quality compiler, and extensive testing is part of the development process. The compiler is verified using an automated test suite with the over 900 test cases of the official AspectJ compiler `ajc`. Many of these cases involve around advice and all of these cases have passed, increasing confidence in the correctness of the strategy and implementation.

We showed how to avoid closure object creation in most cases, and we presented a way to weave around advice in the presence of advice formals with multiple bindings, a case not supported by `ajc`.

Furthermore, we provided benchmarks that show that the around weaver in many cases outperforms `ajc` by up to 200%.

`abc` is available for download at [1].

6.2 Future work

Simple optimizations Possible future work will be to further optimize code generation such as the elimination of unused parameters. Such optimizations can be done purely on the Jimple representation without special knowledge of the weaver. Hence, the preferred approach for this would be to extend the Soot optimizer to perform the necessary steps.

Adaptive inlining An adaptive inliner that inlines around advice as a post processing step where appropriate is another possible optimization.

This optimization is more complicated and it is unlikely to be successful without explicit knowledge from the weaver.

The inlining has to be done in multiple steps. First, the advice method would have to be inlined at the shadow site. After this inlining step, a different optimizer would have to propagate the constant static class ID, and any switch statements that use the static class ID as the switch expression could be eliminated, leaving only the call to the proceed method. Then, the proceed method could be inlined at the shadow site. Based on the constant shadow ID, the switch statement from the proceed method could be eliminated, leaving only the case with the original shadow. Any switch statements and conditionals depending on the constant bind mask could then be eliminated.

Soot provides some inlining functionality, which should be extended and used by the weaver to perform the necessary steps.

Optimization of cycles An interesting point for further investigation would be the optimal ordering of the advice application graph in the case of cycles. In the presence of a cycle, the implementation has to use the closure strategy for one of the advice applications in order to close the cycle. The current application chooses an arbitrary advice application from the cycle. An optimization would be to choose the advice application for which the dynamic residue is least likely to succeed in order to reduce the chance of actual closure creation at runtime.

At compile time, one can clearly distinguish between residues that never fail and residues that may fail, and hence the latter should be the preferred choice for using the closure approach.

A different estimate of the likelihood of success of the residue could be gained by profiling a version of the compiled program. This information could then be fed back to the compiler to create an optimized version of the program.

Bibliography

[1] The AspectBench Compiler. http://abc.comlab.ox.ac.uk/.

[2] AspectJ Eclipse Home. The AspectJ home page. http://eclipse.org/aspectj/, 2003.

[3] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of aspectj programs, 2003.

[4] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[5] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition edition, June 2000.

[6] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[7] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, volume 2622 of *Springer Lecture Notes in Computer Science*, pages 46–60, 2003.

[8] J. Miecnikowski and L. J. Hendren. Decompiling java bytecode: problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer Verlag, 2002.

[9] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.

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

[11] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Foundations of Aspect-Oriented Languages (FOAL), Workshop at AOSD 2002*, Technical Report TR #02-06, pages 1–8. Iowa State University, 2002.

Appendix A

Comprehensive Example

This appendix shows the decompilation of a complete program. The input program is the thread pooling example presented in Section 2.3.4.

Listing A.1: Server

```
1  import java.net.*;
2  import java.io.*;
3
4  public class Server
5  {
6      public static final int PORT_NUM=2748;
7
8      public static void proceed$2(
9                  RequestHandler orgAdviceFormal,
10                 int shadowID,
11                 int bindMask,
12                 java.lang.Object contextArgFormal)
13     {
14         java.lang.Thread requestThread;
15
16         switch (shadowID)
17         {
18             case 0: // the thread start interception
19                 if (bindMask == 1)
20                     requestThread = (Thread)
21                             contextArgFormal;
22                 else
23                     requestThread = orgAdviceFormal;
24
25                 requestThread.start();
26                 return;
27             default:
28                 throw new RuntimeException();
29         }
30     }
```

```
31
32      public static void main(java.lang.String[] args)
33          throws java.lang.Exception, java.io.IOException
34      {
35          java.net.Socket requestSocket;
36          RequestHandler $r1;
37
38          requestSocket = (new ServerSocket(2748)).accept();
39          // thread creation interception
40          $r1 = ThreadPooling.aspectOf().around$1(
41                  requestSocket, null, 0, 1, 0, requestSocket);
42
43          // dynamic residue check,
44          // thread start interception
45          if ($r1 instanceof RequestHandler)
46              ThreadPooling.aspectOf().
47                      around$2(
48                          (RequestHandler) $r1,
49                          null, 0, 1, 0, $r1);
50          else
51              Server.proceed$2(null, 0, 1, $r1);
52      }
53
54      public static RequestHandler proceed$1(
55                  java.net.Socket orgAdviceFormal,
56                  int shadowID,
57                  int bindMask,
58                  java.lang.Object contextArgFormal)
59      {
60          java.net.Socket requestSocket;
61
62          switch (shadowID)
63          {
64              case 0: // request handler creation shadow
65                  if (bindMask == 1)
66                      requestSocket = (Socket)
67                              contextArgFormal;
68                  else
69                      requestSocket = orgAdviceFormal;
70
71                  return new RequestHandler(requestSocket);
72              default:
73                  throw new RuntimeException();
74          }
75      }
76  }
```

Listing A.2: RequestHandler

```
1  import java.net.*;
2
3  class RequestHandler extends java.lang.Thread
4  {
5      private java.net.Socket requestSocket;
6
7      public RequestHandler(java.net.Socket requestSocket)
8      {
9          this.requestSocket = requestSocket;
10     }
11
12     public void run()
13     {
14         // run method replaced by
15         // call to advice method
16         ThreadPooling.aspectOf().
17                     around$0(this, null, 0, 1, 0);
18     }
19
20     // accessor for privileged access
21     public java.net.Socket get$accessor$requestSocket$4()
22     {
23         return requestSocket;
24     }
25     // accessor for privileged access
26     public java.net.Socket set$accessor$requestSocket$5(
27                 java.net.Socket $r1)
28     {
29         requestSocket = $r1;
30         return $r1;
31     }
32     public static void proceed$0(
33                 RequestHandler orgAdviceFormal,
34                 int shadowID,
35                 int bindMask)
36     {
37         RequestHandler this$;
38         switch (shadowID)
39         {
40             case 0:
41                 // the execution shadow
42                 // of the run() method
43
44                 if (bindMask == 1)
45                     this$ = (RequestHandler)
46                             contextArgFormal;
47                 else
48                     this$ = orgAdviceFormal;
```

```
49
50                  PrintWriter writer=null;
51                  try
52                  {
53                      writer=new PrintWriter(
54                          this$.requestSocket.
55                                  getOutputStream());
56                      try {
57                          Thread.sleep(1000);
58                      } catch(InterruptedException e){}
59                      writer.write("Hello!\n");
60                      writer.flush();
61                  }
62                  catch   (IOException ex) {
63                  } finally {
64                      try {
65                          if (writer!=null)
66                              writer.close();
67                          this$.requestSocket.close();
68                      } catch (IOException ex) { }
69                  }
70                  return;
71
72              default:
73                  throw new RuntimeException();
74          }
75      }
76  }
```

Listing A.3: ThreadPooling aspect

```
1  import java.util.*;
2  import java.net.*;
3  import org.aspectj.lang.*;
4  import java.io.*;
5
6  public class ThreadPooling
7  {
8      java.util.Set pool= new HashSet();
9      // the reference to the singleton aspect instance
10     public static final
11         ThreadPooling abc$perSingletonInstance;
12     private static java.lang.Throwable abc$initFailureCause;
13
14     // Thread start advice method
15     public final void around$2(
16         RequestHandler requestHandler,
17         Abc$proceed$ThreadPooling$around$2 closureInterface4,
18         int shadowID5,
19         int staticClassID6,
```

```
20          int bindMask7,
21          java.lang.Object contextArgFormal)
22      {
23          if (requestHandler.isAlive())
24            synchronized {
25              requestHandler.notify();
26            }
27          else
28              // proceed
29              Server.proceed$2(
30                  requestHandler,
31                  shadowID5,
32                  bindMask7,
33                  contextArgFormal);
34      }
35
36      // Request handler advice method
37      public final void around$0(
38          RequestHandler requestHandler,
39          Abc$proceed$ThreadPooling$around$0 closureInterface8,
40          int shadowID9,
41          int staticClassID10,
42          int bindMask11)
43      {
44          while (true)
45          {
46              // proceed
47              RequestHandler.proceed$0(
48                  requestHandler,
49                  shadowID9,
50                  bindMask11);
51
52              requestHandler.set$accessor$requestSocket$5(null);
53              synchronized {
54                pool.add(requestHandler);
55              }
56
57              synchronized {
58                while (requestHandler.
59                    get$accessor$requestSocket$4() == null)
60                {
61                  try {
62                      requestHandler.wait();
63                  }
64                  catch (InterruptedException e){
65                      continue;
66                  }
67              }
68            }
```

```
69          }
70      }
71
72      static
73      {
74          try  {
75                  ThreadPooling.abc$postClinit();
76          }
77          catch (Throwable catchLocal)  {
78              abc$initFailureCause = catchLocal;
79              break ;
80          }
81      }
82
83      // The generated method to
84      // retrieve the aspect instance
85      public static ThreadPooling aspectOf()
86          throws org.aspectj.lang.NoAspectBoundException
87      {
88          ThreadPooling theAspect;
89
90          theAspect = abc$perSingletonInstance;
91
92          if (theAspect == null)
93              throw new NoAspectBoundException(
94                  "ThreadPooling",
95                  abc$initFailureCause);
96          else
97              return theAspect;
98      }
99
100     public static boolean hasAspect()
101     {
102         return abc$perSingletonInstance != null;
103     }
104
105     private static void abc$postClinit()
106     {
107         abc$perSingletonInstance = new ThreadPooling();
108     }
109
110     // Request handler creation advice method
111     public final RequestHandler around$1(
112         java.net.Socket socket,
113         Abc$proceed$ThreadPooling$around$1 closureInterface0,
114         int shadowID1,
115         int staticClassID2,
116         int bindMask3,
117         java.lang.Object contextArgFormal)
```

```
118     {
119         RequestHandler result, result;
120         java.io.PrintStream $r1, $r8;
121         java.lang.StringBuffer $r2, $r9;
122 
123         synchronized {
124           if (pool.isEmpty()) {
125             // proceed
126             result = Server.proceed$1(
127                 socket, shadowID1, bindMask3,
128                 contextArgFormal);
129             return result;
130           }
131           else
132           {
133             result = (RequestHandler) pool.iterator().next();
134             pool.remove(result);
135             result.set$accessor$requestSocket$5(socket);
136             return result;
137           }
138         }
139     }
140 }
```

Listing A.4: Closure interfaces

```
1  public interface Abc$proceed$ThreadPooling$around$0
2  {
3      public abstract void proceed$0(
4          RequestHandler r0, int i1, int i2);
5  }
6  public interface Abc$proceed$ThreadPooling$around$1
7  {
8      public abstract proceed$1(
9          java.net.Socket r0, int i1, int i2,
10         java.lang.Object r3);
11 }
12 public interface Abc$proceed$ThreadPooling$around$2
13 {
14     public abstract void proceed$2(
15         RequestHandler r0, int i1, int i2,
16         java.lang.Object r3);
17 }
```