Time-travel Programming

Programming Language Support for Interacting with Past Executions

by

Robin Salkeld

BMath, University of Waterloo, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

May 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

Time-travel Programming: Programming Language Support for Interacting with Past Executions

submitted by Robin Salkeld in partial fulfillment of the requirements for

the degree of Doctor of Philosophy

in Computer Science

Examining Committee:

Gregor Kiczales
Supervisor

Andrew Warfield
Supervisory Committee Member

Ronald Garcia
Supervisory Committee Member

Norm Hutchinson
University Examiner

Sathish Gopalakrishnan
University Examiner

Additional Supervisory Committee Members:

Supervisory Committee Member

Supervisory Committee Member

Abstract

Because software so often behaves unexpectedly or fails only in production environments, several recent tools from both industry and academia record data about execution for the benefit of post-hoc analysis. Debugging on these data instead of a live program is much more difficult, however, because the semantic abstractions provided by the programming language are no longer available. Many post-hoc analysis tools process this data through additional reflection-based code or domain-specific query languages, but do not recover the expressive power of the original programming language.

This thesis proposes the concept of time-travel programming, which we define as simulating the execution of additional code in the same programming language as if it were present in the past environment of recorded data. Furthermore, we show that the aspect-oriented programming (AOP) paradigm provides a natural mechanism for specifying this additional execution, and allows us to reuse established semantics and implementations. We provide evidence of this technique's flexibility, feasibility and effectiveness through two implementations: one an interpreter for an extremely simple AOP language in the style of a core calculus, and one for the AspectJ programming language. We evaluate flexibility via applying the implementations to multiple execution recording formats, feasibility by showing the AspectJ implementation is performant enough for post-hoc analysis, and effectiveness by demonstrating that evaluating new and existing aspects retroactively can be used to address common post-hoc analysis tasks.

Lay Summary

Just as personal video recorders have enabled recording and replaying live television, several tools support the equivalent of taking a picture of computer software as it runs, or continuously recording it so it can be accurately replayed later. This is potentially invaluable for understanding issues after they happen, but these tools do not yet reproduce the original experience of a developer observing and interacting with live software.

This thesis proposes "time-travel programming" as the concept of evaluating additional code as if it was sent back in time to interact with the original program. We provide the theoretical foundation for this concept as well as a concrete prototype of such a time-travel machine for Java code using three different recording technologies. We demonstrate that this enables powerful after-the-fact analysis by taking existing programming techniques from the present and applying them in the past.

Preface

A version of Chapter 3 has been published as [61]. I was the lead investigator for this research project, responsible for conceptualization, the implementation of the ACC source transformer and runtime based on the Tralfamadore system, and the majority of the manuscript composition. B. Cully, G. Lefebvre and W. Xu contributed various extensions of the Tralfamadore system for the purpose of the runtime and a subset of the text in Section 3.5.2. A. Warfield and G. Kiczales provided high-level guidance and manuscript edits.

A version of Chapter 4 has been published as [59]. I was the lead investigator for this research project, and performed the majority of the design of the RAPL language and the abstract framework for retroactive weaving, the entirety of the RAPL interpreter implementation, and the majority of the manuscript composition. R. Garcia contributed substantially to the design of several language features, and provided manuscript edits.

A version of Section 3.1 and Chapter 5 has been published as [60]. I was the lead investigator for this research project, responsible for the architectural design, implementation, experimental evaluation, and manuscript composition. G. Kiczales provided guidance on methodology and manuscript edits.

The remainder of the dissertation is my own original and unpublished work.

Table of Contents

List of Tables

List of Figures

List of Listings

Glossary

ACC AspeCtC

AOP Aspect-oriented Programming

API Application Programming Interface

AST Abstract Syntax Tree

CDT Eclipse C and C++ Development Tools

CIL C Intermediate Language

CPU Central Processing Unit

DNF Disjunctive Normal Form

DR Deterministic Replay

JDI Java Debugging Interface

JRE Java Runtime Environment

JVM Java Virtual Machine

MAT Eclipse Memory Analyzer Tool

MNM Mirror-based Native Method

PC Program Counter

QEMU Quick Emulator

RAPL Retroactive Aspect Programming Language

TOD Trace-oriented Debugger

TTP Time-travel Programming

VM Virtual Machine

VMM VirtualMachineMirror

XML Extended Markup Language

Acknowledgements

I would like to thank many people for joining me in what has been a long, exhausting, and ultimately rewarding journey.

Andrew Warfield, for encouraging me down this path and providing many entertaining discussions around my crazy ideas.

Ronald Garcia, for providing the fresh perspective I was hoping for.

Gregor Kizcales, for supporting me, for putting up with me, and for teaching me so much more than I ever expected to learn.

Adrian Kuhn, Thomas Fritz, and C. Albert Thompson, for your badly needed doses of encouragement.

My parents, for ultimately making this possible and inspiring me to find out just how much I could accomplish.

My son, for inspiring me to finish what I'd started while simultaneously making it that much harder to.

My wife, for leading by example in every way, for taking it in stride when things were tough, and for believing in me when I couldn't. Above all, this is for you.

A portion of this work was funded by NSERC PGS M and D awards and an NSERC Discovery Grant.

Chapter 1

Introduction

This chapter provides the motivation for the contents of this thesis. It examines how the power of abstraction can be lost when analyzing past executions of software via recorded information. It then introduces the concept of *Time-travel Programming* as a generalized solution to this shortcoming.

1.1 Analyzing Past Executions

One of the most common and basic approaches programmers use to understand execution and diagnose issues is inserting statements that output the state of the program as it executes. Knowledge is built incrementally, as each output may provide clues as to where next to look in the code for more insight and hence where to add the next output statement. Programmers spend many hours in this loop gaining understanding about how code behaves by writing and evaluating more code. As this repetition can be tedious and inefficient, many programming language toolsets include interactive debuggers that support connecting to a running program and inspecting its state. The dynamic evaluation of code snippets in a precise context within a running program is a common feature of such tools, as they mimic the power of modifying and rerunning a program. Thus a programming language is not only the means for humans to tell computers what to do but also a sophisticated means for asking questions about what computers are doing.

An unfortunate side-effect of the ever-increasing complexity of software, how-

ever, is that programs often behave unpredictably when used by real consumers in production environments. Inevitably programmers must figure out why software did not behave as expected in the field, where unexpected behaviour can manifest in many forms: rare or obscure application errors, fatal crashes, exhausting memory or other resources, or simply taking far too long to perform a task.

Ideally developers would like to apply the same interactive, iterative approach above when diagnosing issues that appear in the field, but this is often ineffective since reproducing such issues can be challenging. The same input to a program can produce the correct answer in a development environment and yet fail in a production environment because of subtle differences in contexts. Moreover, some issues will not occur consistently because of the non-determinism due to factors such as concurrency.

In practice, nearly all software will be configured to capture some degree of information about how a program executed so that programmers might have a chance of inferring the cause of unexpected behaviour post-hoc. This commonly takes the form of logging statements that capture some of the human-readable information described above. Programmers can diagnose some bugs and unexpected behaviours simply by reading the high-level, vastly simplified timeline of a program's execution contained within its logs. They allow programmers to become computing archaeologists, sifting through artifacts from the past in order to glean insight.

For deeper and more sophisticated debugging and runtime analysis the manual approach does not scale, so it becomes necessary to automate the analysis. Thus the recorded information needs to be machine-readable rather than human-readable. Many mainstream programming environments can produce a snapshot of a program's state at a single instant in time for later analysis, either on request or as an automatic reaction to encountering fatal errors. Moreover, technology for recording the continuous execution of a program is an active area of research [18, 49, 51, 62], and is becoming efficient enough to find its way into mainstream computing environments [63].

However, the information in any such execution recording is encoded as low-level data and events in the recorded programming language runtime, or even in the operating system or hardware beneath it. The sophisticated layers of abstraction in

the program have been lost, leaving a substantial semantic gap. Related tools and research take the approach of building libraries of operations to analyze or visualize common datatypes from the recorded data in order to enable insight. The datatypes and operations will be represented by a user-level application model: either expressed within some other programming language, sometimes a specialized query language, or in the original language but at the metaprogramming level. Because these techniques do not support evaluating code in the original programming language, they are forced to manually reproduce the behaviour of a live runtime, which is difficult and time-consuming to accomplish for even a small subset of commonly-used code and impossible to generalize to all possible programs.

1.2 Time-travel Programming

The key inquiry motivating this thesis is to what extent it is possible to reuse both the semantics of and the code written in a programming language to interact with its own execution recordings. For this to be useful for software developers, the semantics of the relevant code must be maintained: any retroactive computation must be consistent with how the execution would have behaved if it had been present during the original computation. We define *Time-travel Programming (*TTP*)* as the paradigm of programming in the context of a recorded prior execution, as if new computation is being sent through a time machine to occur in the past execution context.

We refer to this simulated execution as *Retroactive Execution*. Its defining characteristic is consistency with the preexisting semantics of the programming language. In the simplest case, the semantics of evaluating an expression such as `print(tree)` requires a great deal of context. Providing a sound interpretation of the programming language's semantics is equivalent to recreating the behaviour of a debugging tool connected to a live program. Therefore, in general, implementing retroactive execution involves simulating the entire language runtime.

Execution recordings such as those described in Section 1.1 are always incomplete to some degree for any programming language that supports interacting with anything outside the program's internal state. In any realistic execution environment there will be factors that are implicitly omitted from recordings, such as the

exact timing of events, external resources such as file systems and networks, nondeterminism in the presence of concurrency, and so on. In addition, recordings will often be explicitly scoped in order to keep performance and recording size manageable. If retroactive execution attempts to access any of this missing data, there is no guarantee that the simulation will behave consistently as TTP requires. In these cases the simulation must fail, and hence retroactive execution implementations by definition will always be partial.

Many forms of analysis require coordinating multiple such additional computations over time as it is represented in an execution recording. For example, if a bug in a program's implementation caused a binary search tree to mysteriously lose an element the tree could be inspected repeatedly over time to pin down exactly when it happened. This amounts to simulating the behaviour of adding additional code at multiple locations in the source. Specifying this additional, potentially complex and stateful computation as it relates to the original program is a natural programming language design problem, and is best solved by extending the original programming language while maintaining source-level compatibility.

We observe that this additional computation will by necessity crosscut the original computation, and hence that this problem aligns precisely with Aspect-oriented Programming (AOP). Casting time-travel programming into the AOP paradigm allows us to leverage not only the established semantics of AOP languages but also their implementations. We define *Retroactive Weaving* as the process of evaluating one aspect of a program with respect to the prior execution of another aspect, in contrast to conventional weaving as illustrated in Figure 1.1. Retroactive weaving then becomes an alternate implementation strategy for providing the established semantics of the AOP language, changing not what the aspect means but when it is actually executed.

1.3 Thesis Statement

This thesis demonstrates the advantages of time-travel programming: reusing a program's source code and its programming language, or an extended version of it, to analyze and interpret its own prior computation. We show that this generalized technique is a flexible, feasible and effective paradigm for providing post-hoc

runtime analysis.

1.4 Claims and Contributions

We support our thesis statement with three key claims:

- TTP is effective, in that it supports the natural and succinct expression of many post-hoc analyses;
- TTP is flexible, as it is applicable to any programming language and any technology for recording that programming language; and
- TTP is feasible for a mature mainstream programming language, in that:
  - it can be implemented by reusing the language's original implementation;
  - it can achieve performance adequate for the purposes of post-hoc analysis; and
  - it can succeed frequently enough to be useful despite incomplete information about the original execution.

We provide evidence to support these claims through the following contributions:

- An abstract formal framework for retroactive weaving as it relates to AOP, thus defining the core semantics and requirements of TTP;
- A discussion of the design space for a valid TTP implementation via an interpreter for a simple core language with TTP features, demonstrating the flexibility and generality of TTP;
- An architecture for and a library implementation of TTP for the Java Virtual Machine (JVM) using the AspectJ AOP language and reusing its implementation, further demonstrating the flexibility of TTP via pluggable implementations for three distinct execution recording technologies; and

- An evaluation of the effectiveness of the JVM implementation at addressing common post-hoc runtime analysis problems in terms of its runtime performance, ability to reuse existing code, and success rate of producing useful results despite incomplete information, thus providing evidence of:
  - The effectiveness of TTP via useful examples in AspectJ; and
  - The practical feasibility of TTP for a mature mainstream programming language.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides information on existing concepts and prior work that this thesis builds on. Section 1.2 discusses the motivation for the concepts this dissertation proposes and a prototype implementation of those concepts. Chapter 4 provides a foundation for the semantics of retroactive aspects by applying the concepts to a simple core language. Chapter 5 then focusses on an implementation and evaluation of retroactive execution, a related concept necessary to support retroactive weaving but with a large amount of complexity and independent utility of its own. Chapter 6 then presents and evaluates a retroactive weaver for AspectJ based on extending this implementation. Finally, Chapter 7 summarizes the dissertation and examines the limitations of this research and potential avenues for future work.

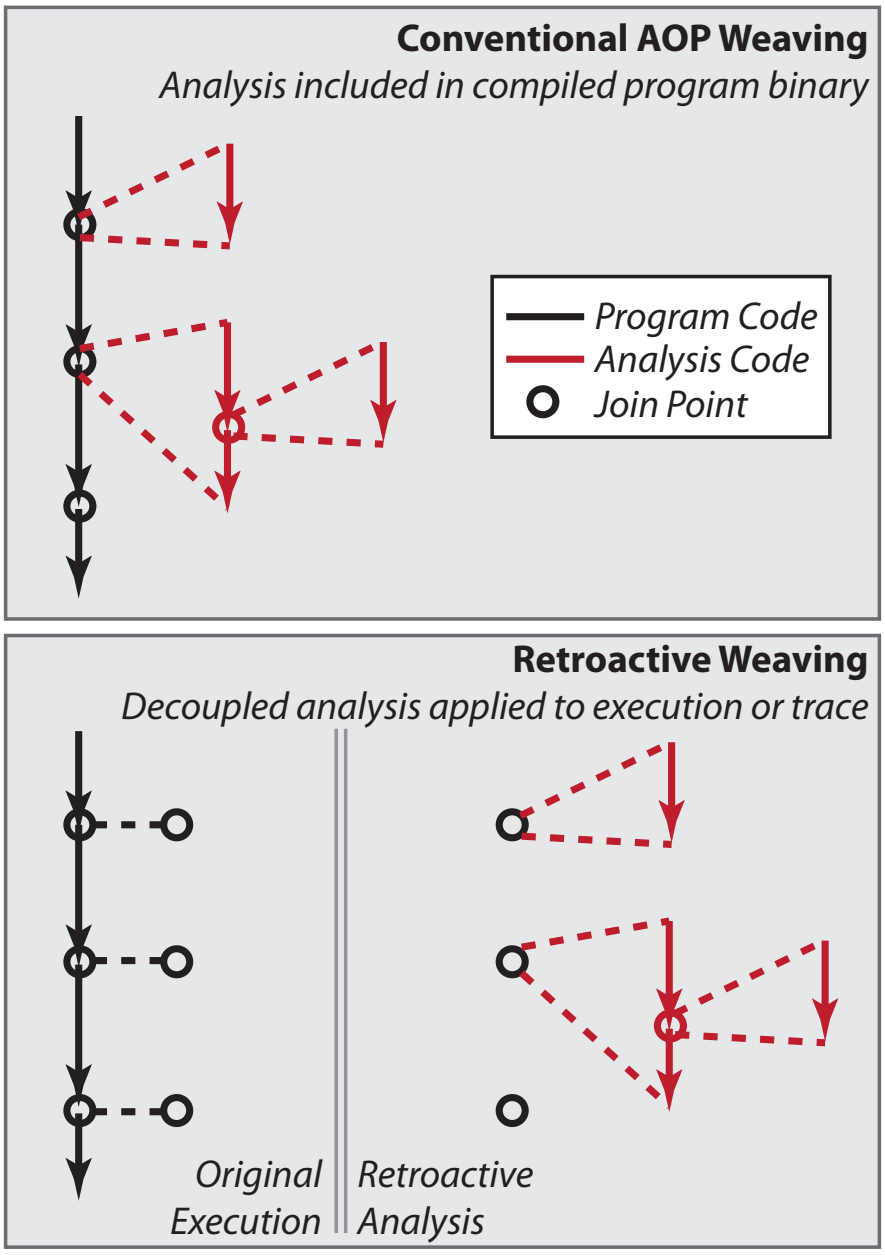

Figure 1.1: Traditional aspect weaving versus retroactive aspect weaving

Chapter 2

Background

The concepts we introduce in this thesis are synthetic and borrow from multiple mature fields of research. This chapter provides important background information on those fields and relates prior work to the work described in the following chapters.

2.1 Complexity and Abstraction

As computing becomes more and more omnipresent in our lives, software continues to grow more and more complex. This complexity is made manageable by layers of abstraction. Programming languages abstract the low-level details of the hardware they run on, and datatypes abstract the implementation of the high-level concepts they represent.

Consider the binary search tree, one of the simplest common data structures in computer science. Semantically, this abstract datatype represents an unordered set of elements. It is implemented as a binary tree where the element in each node is greater than all elements in its left subtree and lesser than all elements in its right subtree. Figure 2.1 presents a sample binary tree containing five integers. Typically the binary search tree implementation would be provided as a reusable library and include a collection of functions for querying and modifying binary search trees. Applications are composed of many such abstractions, often layered on top of each other.

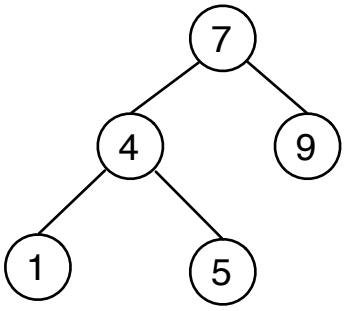

Figure 2.1: A sample binary search tree, representing the set $\{1,4,5,7,9\}$

...	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
...	7	17	11	5	0	0	9	0	0	1	0	0	4	14	8	...

Table 2.1: A possible memory layout of the sample binary search tree

At a lower level, the nodes of the tree structure itself are usually implemented as non-contiguous chunks of computer memory. Each cell contains three pieces of data: the addresses of its two subtrees, left and right; and the actual value stored in that cell. Table 2.1 illustrates a simplified view of a possible memory layout for the sample binary search tree in Figure 2.1. The upper row contains the address of each memory location and the lower row contains the actual value stored at that location. Manually reading the semantic contents of this data structure based on this memory is challenging, especially when in practice the size of such a binary search tree is frequently on the order of hundreds or thousands of elements.

It is straightforward, though not completely trivial, for the binary search tree to provide a function that produces the human-readable representation of its contents. The code in such a function would traverse through the nodes of the tree in order, most likely using recursion, incrementally constructing a string of characters. Given the tree in Figure 2.1, it would produce the string "$\{1,4,5,7,9\}$".

This kind of functionality is extremely important to programmers: it enables them to reason about code using the abstraction it represents instead of the complexity it hides. When attempting to understand a large software project or determine the root cause of a subtle bug, one of the most common and effective

approaches is to add more code that outputs such human-readable encodings of a program's state at various points as it runs, or to connect to a running program with a debugging tool that supports interactive evaluation of code snippets.

Thus programming is not a fire-and-forget exercise. A significant portion of the process of writing software is interactive and incremental. Programmers spend many hours in the code-evaluate-repeat loop, gaining understanding about how code behaves by writing and evaluating more code. A programming language is not only the means for humans to tell computers what to do, but also a means for asking questions about what computers are doing.

2.2 Reflection and Metaprogramming

The term *metaprogramming* refers to manipulating programs or their runtimes. Compilers and translators are hence a kind of metaprogram, since they refer to code as data. *Reflection* refers to elements of a programming language that reference its own code or data, and hence is a specialization of metaprogramming targeted at the same program [64]. Reflection adds another level of expressive power and flexibility to a programming language, allowing programs to generalize over otherwise unrelated elements such as fields of an object.

Java's reflective interface consists of several closely-related classes in the core library such as `Class` and `Field`. These classes provide several methods which reflect on the state of the runtime, including classes, objects and threads. To read the value of the field `f` on the object `o`, for example, the straightforward or "base-level" expression would be `o.f`. The same value can be retrieved using reflection, assuming `o` is an instance of the class `c`, using the expression `c.getDeclaredField("f").get(o)`. A reflective interface adds a level of indirection and provides greater flexibility, but also requires a more verbose syntax than the base level code.

Kiczales et. al. [33], in describing metaobject protocols, provide terms for three distinct levels of self-reference of increasing power, which we use in reference to metaprogramming in general: *introspection* refers to read-only access to the state and structure of a program, *modification* (originally *self-modification* in the reflective case) refers to modifying this state reflectively, and *intercession* refers to

modifying the behaviour of elements of the program. Java's reflective interface offers a substantial set of reflective operations for introspection but relatively few for self-modification or intercession: it is possible to change field values, for example, but not to change the definition of already-defined methods.

2.3 Debugging

The term debugging in a broad sense is the process of software developers analyzing the cause of undesired behaviour in their code. For the purposes of this thesis, we use the term *debugging* to refer more specifically to interacting with a program as it runs, in order to extract additional information about it or even to make changes on the fly in order to test potential bug fixes.

Debugging tools are built on another example of metaprogramming, as they connect to a running program in order to examine and in some cases alter the state of that program. The Java Debugging Interface (JDI) is one example of an Application Programming Interface (API) for interacting with the state of a running program from within a debugger tool. The JDI includes interfaces that resemble the native Java reflective classes superficially: `ReferenceType`, for example, is the equivalent of `Class`, and both define a `Field` type. The JDI offers a metaprogramming interface that is strictly more powerful than native Java reflection. It is possible, for example, to force the currently active method invocation to return early with a given value.

To support human interaction in debugging tools, the JDI also includes a rich event-driven API. In particular, it is possible to describe events of interest, such as when a particular method is invoked, and receive notification when those events occur. While clients of the JDI are handling such events the thread responsible for raising the requested event is suspended, or even the entire target program paused if the clients request it. This capability enables a limited form of intercession, as the JDI client can intercept the behaviour of certain program elements and modify program state before resuming.

Figure 2.2 shows an example of the Eclipse user interface, which is built using the JDI, while debugging a Java program. The lower pane displays the output of the program so far, and the middle pane shows the line of code currently executing.

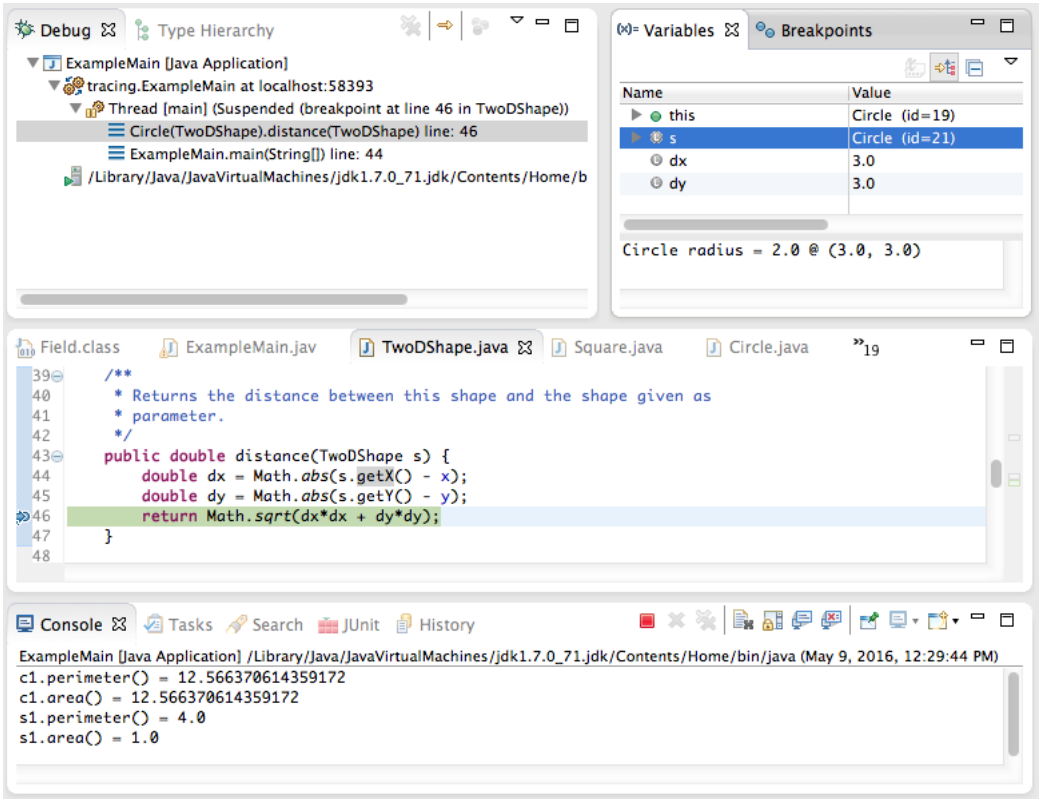

Figure 2.2: An example of the Eclipse user interface while debugging a Java program

The small circle on the left indicates the user has set a breakpoint at the current line, which caused the program to pause when it reached this line. The upper-left pane displays active threads and their state, in particular their current stack of method invocations. The upper-right pane lists the symbols that are in scope in the current context and their values. Note that because the variable `s` is selected in this pane, the result of invoking `s.toString()` is displayed just beneath the list of symbols.

In general, the trade-off for offering the connectivity and metaprogramming features for debugging described above, in particular monitoring the program for requested events, is to significantly slow down the programming language runtime. Therefore, such runtimes typically provide a so-called debugging mode which must

be explicitly turned on and is usually left off for production.

2.4 Aspect-oriented Programming

The paradigm known as *aspect-oriented programming* (AOP) was inspired by the observation that programs contain many concerns that are orthogonal to the traditional decomposition of functions or class methods. Software design typically includes several such concerns, such as logging, verification, security, persistence, and event handling. Concerns such as these are said to *crosscut* each other if their areas of effect in a program intersect but are not strictly contained in each other. In non-AOP languages this orthogonality forces the code that implements each concern to be scattered throughout the code, leading to duplication, increased maintenance cost, and decreased developer understanding of the system's design. AOP languages include one or more features designed to allow crosscutting concerns to be implemented as independent modules. See Section 2.5 for a concrete example.

Early presentations of the aspect-oriented programming paradigm focussed on decomposing a program into a *base program* and one or more *aspects*, where the base program was always modularized along the traditional dimension of functions or class methods. Masuhara and Kiczales [43] later provided one of the more generalized definitions of cross-cutting modules and of aspect-oriented programming, and we adopt their terminology here. Their model defines *weaving* as the process of coordinating the cross-cutting specifications of two independent programs to produce the behaviour of the combined program. At a high-level, weaving is a function with signature[1] $A \times B \rightarrow X$, where A, B and X are all languages, not necessarily distinct from each other. The term *join point* refers to individual elements of the combined program where the two input programs intersect. A and B both include mechanisms to identify relevant join points in the X program and how they each affect the semantics of those join points. There are several broad classes of weaving implementations and many strategies for efficient dispatch, but this model describes the semantic requirements on such implementations.

[1] Masuhara and Kiczales' version also includes a third optional *META* meta-language used to allow external configuration of how the two programs are combined. This is not relevant for the content of this thesis and we omit it here.

2.5 AspectJ

AspectJ is an aspect-oriented extension of Java, and is one of the most well-known AOP languages. It adds `aspect` definitions, which are somewhat similar to classes but are designed to support modular implementations of concerns that typically crosscut class and method decompositions. Aspects contain *advice* declarations, which are similar to methods but are executed in response to relevant program events rather than explicitly invoked. *Pointcuts* are declarative expressions which pick out these sets of relevant program events. Aspects can also contain *intertype definitions*, which semantically add additional static type structure to one or more unrelated classes, such as declaring additional fields, methods, or implemented interfaces.

A frequently-cited application of aspect-oriented programming is tracing program execution, which in practical terms means outputting relevant information when key events occur. Below is a simplified version of this aspect as packaged with the Eclipse-based distribution of AspectJ and described in the AspectJ Programming Guide [2]:

```
public class Trace {
  protected static PrintStream stream = System.err;
  protected static int callDepth = 0;

  public static void traceEntry(String str, Object o) {
    callDepth++;
    printIndent();
    stream.println("--> " + str + ": " + o.toString());
  }

  public static void traceExit(String str, Object o) {
    printIndent();
    stream.println("<-- " + str + ": " + o.toString());
    callDepth--;
  }

  private static void printIndent() {
    for (int i = 0; i < callDepth; i++) {
      stream.print("  ");
    }
```

```
  }
}

aspect TraceMyClasses {
  pointcut myClass(Object obj):
      this(obj) && within(TwoDShape) || within(Circle) || within(Square);
  pointcut myConstructor(Object obj): myClass(obj) && execution(new(..));
  pointcut myMethod(Object obj): myClass(obj) &&
      execution(* *(..)) && !cflow(execution(String toString()));

  before(Object obj): myConstructor(obj) {
    Trace.traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
  }
  after(Object obj): myConstructor(obj) {
    Trace.traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
  }
  before(Object obj): myMethod(obj) {
    Trace.traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
  }
  after(Object obj): myMethod(obj) {
    Trace.traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
  }
}
```

The `myClass` pointcut matches all join points within the code that defines the three listed classes. The `myConstructor` pointcut intersects this pointcut with one that matches the execution of any constructor, and hence matches all constructor executions in these classes. This aspect therefore causes the execution of the tracing statements before and after every constructor or method execution in the listed classes. In a non-AOP implementation, these calls to the tracing module would have to be duplicated in every constructor and method in these classes. Note that the `cflow` clause in the `myMethod` pointcut is a common idiom used by advice to exclude the join points in the control flow of their own code and hence avoid infinite regress.

The AspectJ distribution includes the de facto standard implementation of AspectJ, which we will refer to as `ajc`. This toolset implements the semantics of aspects by modifying the code that is run, as described in [30]. The AspectJ compiler and runtime define a common binary format for aspects. Each aspect is compiled

into a Java class, and each advice definition becomes a method (which we refer to as an *advice method*) with a generated internal name and annotations containing information relevant for weaving, such as the associated pointcut definition. Other auxiliary data such as `cflow` state data structures are inserted into these classes as needed.

The elements of code that are causally linked to join points at runtime, and hence need to be augmented in this way to implement the cross-cutting nature of AOP programs, are referred to as *join point shadows*. Each join point shadow is matched statically with each pointcut, and wherever there is a potential dynamic match, the shadow is altered to implement the advice semantics as well. The exact implementation details vary, and may involve inserting calls to generated methods containing advice bodies, or inlining advice bodies directly with shadows. Similarly, intertype declarations are generally implemented by inlining the declarations into their target type.

`ajc` supports two approaches to weaving, both of which use this binary aspect format at runtime. *Compile-time weaving* locates the weaving process in the compilation stage. All code, including that contained by either methods or advice definitions, is augmented as it is compiled to JVM bytecode to include calls to the munged advice methods as needed. *Load-time weaving*, by contrast, applies transformations to JVM bytecode as it is loaded into the JVM and before it is executed.

Because of their very cross-cutting nature, the scope of code that aspects affect is not manifest in aspect source or binaries themselves. This is instead determined by external configuration: compile-time weaving is scoped by the build environment, and load-time weaving by configuring the Java runtime to affect how classes are loaded, generally by creating custom class loaders.

Other AOP languages support what is called *runtime weaving*, which is applying the semantics of aspects dynamically. The definition of the AspectJ language does not support runtime weaving: among other reasons, the semantics of disabling an aspect containing intertype definitions that are referenced by running code are unclear. However, dynamic weaving is a well-established approach in the literature [9] [54] as well the preferred method for dynamic AOP languages [26].

From the early days of research on aspect-oriented programming, many have observed when a program is factored into multiple aspects, it is natural to classify

those aspects as either interfering or non-interferring. The latter style of aspects are in some sense safe to omit as needed, such as to increase performance in production environments.

One of the often-cited benefits of AOP is flexibility. An aspect that traces a large volume of debugging information may be easily turned off when deploying though a small amount of compiler or runtime configuration. Any such aspect that is not strictly required for program correctness is certainly optional and non-interferring in some sense. Dantas and Walker [22] offer a formal definition of harmless advice which prevents such advice from altering a computation's final value, while still allowing the advice to read state of interest and perform input from and output to the outside world.

2.6 Execution Recording

This section describes a number of variations on the theme of recording some subset of the state of a program as it executions.

2.6.1 Snapshots

We use the term *snapshot* to refer to any persisted record of the complete state of a running program to a high degree of detail. Many operating systems automatically produce a core dump when a program crashes, which is one form of snapshot that records the low-level state of a program's raw memory. It may also include the contents of Central Processing Unit (CPU) registers.

A JVM will produce a so-called *heap dump* when it exhausts available memory. A heap dump records the state of a Java program in terms of Java semantics, as objects and primitive values and the references between them rather than raw memory state. It does not, however, include the Program Counter (PC) counter of each thread of execution.

2.6.2 Traces

The idea of recording snapshots can be semantically extended to record the state of a program over time as it executes, rather than at a single instant in time. Rather than recording multiple independent snapshots, however, this can be done much

more efficiently by only recording the incremental changes between timestamps. Indexing these deltas enables efficient handling of requests for specific runtime values at specific times.

Omniscient debugging [38] describes narrowing the gap between live debugging and execution trace analysis by creating trace browsers that look like live debuggers. Developers can examine variable values and control flow state using the same idioms as live debugging, but also travel backwards and forwards through time to events of interest. Research has largely focused on improving the execution tracing and trace querying efficiency [55, 56] of omniscient debugging.

2.6.3 Deterministic Replay

Deterministic Replay (DR) [24, 25, 31, 68] is an active area of research that enables recreating the actual execution of a program exactly as it previously happened. This is accomplished by recording sources of non-determinism in the program as it executes. This includes external sources such as user input, but also internal sources such as the results of concurrent access to shared data. A later replay stage reads this recording and forces the same outcome of non-deterministic operations as before.

Chapter 3

Motivation

This chapter identifies some of the shortcomings of the various systems available for post-hoc dynamic analysis, and argues for the position that time-travel programming is a natural and effective solution. It also describes an initial implementation of retroactive weaving for AspeCtC (ACC), an aspect-oriented extension of C, using static analysis and code transformation techniques. This implementation provided some evidence for the effectiveness of our position although it was not developed further. We then discuss the positive and negative observations made in our experience with this prototype.

3.1 Compatibility with Program Source

We first describe the approach many post-hoc analysis tools use to work around the inability to execute code, which is to force developers to program analysis using a reflective model, and examine the problems we intend to solve with retroactive execution.

For the purposes of concrete discussion we will first focus on Java heap dumps as a particular example of recording execution; as outlined in Section 2.6 such snapshots of execution state are a key part of many approaches to recording execution. There are many types of retroactive analyses that can be applied to a snapshot: counting the number of objects of each unique type and how much memory they occupy, for example, to determine the root cause of exhausting available mem-

ory. The Eclipse Memory Analyzer Tool (MAT) [3], which we use to represent the current state-of-the-art, includes a rich UI for browsing the object graph, several dozen actions for navigating it, and several pre-defined analysis commands such as identifying the set of objects that are most likely leaking memory.

Consider a Java developer browsing the object graph in a heap dump who has identified an object that implements the built-in `java.util.Map` interface. This implies that the object implements an associative mapping from keys to values. Suppose the developer needs to know what value a certain key is mapped to. When debugging a live process this is as simple as evaluating the expression `map.get(key)`, but the task is surprisingly involved with a dead process. Even if the concrete type of the map was `java.util.HashMap`, a well-known implementation backed by a straightforward hash table, finding the right key-value pair involves manually searching through the internal representation of a potentially massive hash table to find the matching entry.

Not being able to execute code also means there is also no way to invoke the method `Object.toString`. This method is defined at the top of the Java class hierarchy with a default implementation and therefore callable on any object, and is used to create human-readable text describing the object. The Eclipse debugging perspective, for example, includes a window dedicated to displaying the result of calling `toString` on objects in other windows (as shown in Figure 2.2), and it is exactly this basic functionality that is missing in a heap dump browser.

The tools used to interact with heap dumps attempt to address this issue by providing utilities to automate these tasks: in particular, the MAT includes a command named "Extract Hash Entries". To implement this utility, however, the tool developer must include handler code for every single subclass of `java.util.Map` that might be encountered in the object graph.[1] Since the tool cannot possibly handle all possible types in arbitrary application code, the utility must be pluggable so that developers wishing to analyze their heap dumps can contribute handlers for the types of objects in them. Ultimately, this means that those developers are faced with the job of re-implementing the semantics of a non-trivial portion of application code using the meta-level represented by the heap dump model.

[1] The standard library alone includes several dozen, such as `ConcurrentHashMap`, `LinkedHashMap`, `TreeMap`, and so on.

```
1  public String getProperty(ServiceProperties map, String key) {
2      return (String)map.get(key);
3  }
```

Figure 3.1: An example of working with an associative mapping using application code.

```
1  public String getPropertyMeta(IObject map, String key) {
2      String[] keys = null;
3      String[] values = null;
4
5      IObject object = map.resolveValue("headers");
6      IObjectArray array = (IObjectArray)headers;
7      if (array != null) {
8          long[] keyAddrs =
9                  array.getReferenceArray();
10         if (keyAddrs != null) {
11             // Call helper method to dereference
12             // addresses to String objects
13             keys = getServiceProperties(keyAddrs);
14         }
15     }
16
17     /* Similarly for "values" field */
18
19     if (keys == null || values == null)
20         return null;
21     for (int i = 0; i < keys.length; i++) {
22         if (keys[i].equals(key)) {
23             return values[i];
24         }
25     }
26     return null;
27 }
```

Figure 3.2: The same example as in Figure 3.1, but using the meta-level interface of a heap dump model

Figure 3.1 contains an example of base code that retrieves the value a key is mapped to in one particular type of map, and Figure 3.2 contains an example of MAT source code that achieves same thing using the reflective heap dump API. This snippet is part of a large submodule responsible for reconstructing the state of the Equinox OSGi Java module system [29] from a heap dump; this code is only concerned with the configuration properties, a mapping from strings to strings, for a particular service. The configuration is stored as a pair of parallel arrays, and this code first resolves the object references contained in each, then iterates through the list of keys until it finds a match. This example illustrates a number of disadvantages inherent to this approach.

Complex: The complexity inherent in reproducing the behaviour of a polymorphic method such as `Map.get` is high. As shown in Section 2.2, meta-level code is inevitably much more verbose than the equivalent base-level code, and reproducing higher-level language features such as polymorphic method dispatch and field shadowing correctly at this level is challenging.

Type-unsafe: If the analysis code makes a type error, it will not cause an error until further downstream reads fail to find the fields they are expecting, if an error occurs at all. This makes tracking down coding errors much more difficult. In Figure 3.2 the meta-level code assumes that the property values are strings when in fact they can be other types, and in the base-level code a cast is necessary to make the code compile. Guarding against such errors through explicit meta-level type checks requires programmer discipline, contributes greatly to code bulk, and as shown in the example are often omitted.

Brittle: In Figure 3.2, the code expects the properties to be stored in two parallel arrays. In more recent versions of Equinox, however, these fields no longer exist, and the properties are instead stored in a single special-purpose object. Because of the various null checks in this code, likely present to prevent this exact problem from crashing the overall analysis, no error is raised, and the service object appears to have no properties. The fact that meta-level analysis accesses datatype internals directly means that the analysis can very easily break or, even worse, silently produce incorrect results when run against a dump produced by a newer version of the target code. Since this approach involves a second, redundant implementation of the functionality in the application, it introduces an ongoing maintenance burden.

Insecure: The access control mechanisms built into a language, such as declaring types or fields `private` or `protected` in Java, are generally more easily circumvented within its own reflection facilities. Access control depends on knowing the accessing context, which for Java means the class the referencing code resides in. The dynamic nature of reflection means that this is not statically available, and not even dynamically available unless explicitly provided at great inconvenience. When traversing a heap dump model these protections are lost entirely: in the example above the code reads from two private fields with no extra difficulty. This contributes towards the brittleness of analysis code as above, since it is easy to refer to internal details that are likely to change in the future, but it also means that sensitive data is easy to read, whether by accident or on purpose.

Unfamiliar: Developers familiar with base-level (ordinary) programming in a language must learn an additional paradigm to understand and effectively work with the meta-level object model. In addition, because application values frequently refer to datatypes from libraries, reimplementing application-level code requires understanding and reimplementing library code as well, since the encapsulation that normally hides those details from clients has been lost. Tool developers can alleviate some of this burden by handling the most common functionality, but handling even a small percentage of the code a user is likely to encounter is impractical.

Most debugging and analysis tools that are capable of reading snapshots or core dumps share these limitations. The GNU Project Debugger (GDB), for example, supports automated debugging through Python scripts, but these scripts operate on a similar meta-level representation of program state.

The primary benefit of retroactive execution as a tool for retroactive analysis is to provide the same analytic power as the original programming language, enabling the developer to simply evaluate `map.get(key)` in the context of the snapshot and obtain the needed answer.

3.2 Coordination with Past Execution

The previous section provided motivation for the utility of retroactive execution. Automated post-hoc analysis code, however, also needs to specify individual points

in time of interest in the program's execution to trigger additional code, and have access to local values that aren't available from the global state. Retroactive execution restores the expressive power of code for interacting with a single point in time, but does not by itself provide an equally expressive mechanism for interacting with the flow of state over time. Without such a mechanism, locating relevant points in time and collating or comparing values over time is just as complicated and error-prone as the metaprogramming-based alternative to retroactive execution outlined above.

Ideally, offline analysis should be just as intuitive and familiar as adding `printf()` statements within a program's source code: the analysis' interaction with the original program should be type-safe; consistent with its value, name binding and control flow semantics; and able to reuse its datatypes and algorithms. In addition, it should be possible to author analysis code with consistent semantics that do not depend on which execution capture and replay technology is used, or even on whether the analysis is evaluated during the program execution or after it.

Offering the same flexibility as inserting additional code at arbitrary locations in the source code is challenging, and ensuring the instrumentation is run exactly when needed often involves complicated and flow-sensitive specifications. This problem is addressed in various ad-hoc ways by existing dynamic analysis tools, both online or offline. Pin [41] and the ATOM language it is based on [65] are quite aspect-like at the instruction level, but each instruction must be individually inspected as a metadata object to determine where to insert additional instructions. The Program Trace Query Language (PTQL) [28] is a reduced dialect of SQL customized to query program executions, which is described as being equally applicable to post hoc evaluation. PTQL treats individual execution events such as function calls as relational tuples, with time as an explicit dimension that must be manually manipulated, which makes analysis of control flow awkward to specify. VMWare's VAssert [4] system supports replay-only statements in source code, which requires analysis foresight and is less convenient for testing bug theories after the fact.

Coordinating the interaction of the cross-cutting analysis with the original program execution lies precisely in the domain of aspect-oriented programming. In particular, pointcuts provide a rich specification language for solving exactly this

problem, and are equally applicable to matching joinpoints that occurred in the past. They include binding mechanisms such as `args(x, y)` and `return(z)` which allow values to be extracted from the original program and used in analysis code. Using a declarative sub-language to describe analysis trigger points also presents opportunities for optimization in the context of execution tracing and replay, since irrelevant events and even entire periods of execution can be ignored or skipped over when reading a trace or instrumenting a dynamic replay process.

In some cases an AOP language's pointcut types may not express all of the desired concepts for post hoc analysis, but as proposed previously [12, 20] it is reasonable to extend pointcut languages for greater expressiveness or precision; we have extended ACC ourselves, in fact, in order to address its shortcomings when applied to kernel-level code, as described in Section 3.5.2. Implementing AOP concepts beyond pointcuts and advice in the context of retroactive evaluation could also enable more sophisticated analysis. Intertype declarations, in particular, offer an attractive solution for attaching additional analysis state to program structures after the fact, and are flagged as future work in our prototype.

3.3 Access to Program State

Retroactive aspects will read values from the original program execution wherever such values are referenced through pointcut arguments or global variables. Retroactive weaving therefore needs to extract those values from the original execution.

Simply applying traditional dynamic advice weaving to a deterministic replay process would appear to be an ideal implementation, but unfortunately this will inevitably interfere with the fragile replay process. In our case of kernel-level process replay, for example, it is necessary to record all system calls made to the kernel so they they can be reproduced in replay. Any system calls made by the analysis code will be incorrectly trapped by the replay process and produce the wrong results, and in addition throw off future results, so even an apparently harmless call to `printf()` can be disastrous. Similar issues exist for any replay mechanism: in general, efficient replay relies on being able to assume all deterministic behaviour remains exactly the same between record and replay. Post hoc analysis must there-

fore be somehow isolated from the DR process it analyzes.

Recently, Devecsery et. al. showed in their work on *eidetic systems* [24] that it was feasible to record all user processes in a system with minimal overhead and storage on the order of a few terabytes over several years. They also leveraged prior techniques [7] for ensuring their Pin-based analysis does not perturb their DR processes: ensuring the memory allocated by analysis does not overlap with the original memory referenced by the recorded process, and distinguishing system calls made as part of replay versus those made during analysis code. This does mean, however, that analysis must be written to explicitly copy values into this isolated memory space in order to analyze them, and if analysis code invokes existing code in the process it runs the risk of leaking references to this space and invalidating the isolation.

Because retroactive aspects are compatible with the program source and can maintain their own state, references to both the original program's state and the analysis state must be disambiguated. In our C implementation, for example, this means that pointer values may actually refer to either the original program's memory or the new analysis runtime's memory. We refer to this as the *dual address space problem*.

In other decoupled analysis frameworks such as Speck [49], data must be manually marshalled between processes in order to synchronize them, which does not scale well to arbitrary source-level analysis. Choi and Alpern use *remote reflection* [48] to debug a Jalapeo JVM [17] from another JVM. Remote reflection solves the dual address space problem in Java dynamically by customizing the debugging JVM to track whether each object is remote or local; references to remote object members are handled by communicating with the JVM being debugged, and the flag is propagated through such references.

Our approach in our retroactive weaver for ACC is to solve the problem statically instead. At compile time, variables declared by the program source or bound by pointcuts are marked by our aspect compiler as program state. Other variables and fields are inferred based on dataflow as either program or analysis state within the source code, and references to program state are rewritten to instead call a runtime API to recreate that state.

This concept can be illustrated with a simple example: Figure 3.3 contains a

```
1  before(struct mutex * lock) :
2          call($ mutex_lock(...)) && args(lock) {
3    if (!check_order(current, lock)) {
4      printf("Lock order reversal: %p (%s)", lock, lock->name);
5    }
6  }
7  before(struct mutex * lock) :
8          call($ mutex_unlock(...)) && args(lock) {
9    remove_from_order(current, lock);
10 }
```

Figure 3.3: A simple ACC example inspired by a Linux kernel module

sample ACC aspect inspired by the `witness` kernel module, which validates that lock acquisition and release code is structured correctly to avoid deadlocks by verifying that lock ordering is consistent. The `check_order` and `remove_from_order` functions manipulate a lock order relationship tree; their implementation is omitted for brevity. This is one of several units of conditionally-enabled kernel code that provides valuable debugging information but is normally too slow or intrusive to enable for production builds.

In this aspect, the `lock` pointer is bound though the `args` pointcut, and is hence marked as referring to base state, as is `lock->name`. So is the result of `current`, a Linux kernel macro used to get a pointer to the currently running task from a CPU local variable. The string literal in the call to `printf`, on the other hand, is a new value created by the aspect code and hence its address is marked as referring to aspect state. Our implementation of this inference process is described in Section 3.5.1.

3.4 Avoidance of Side-Effects

For source-level post hoc analysis to have consistent semantics whether it is evaluated online or offline, the analysis code must not have side-effects that would have changed the program behaviour; the analysis framework should reject any such analysis.

In other systems where analysis is performed outside the original program, such as Speck or remote debugging, any side-effects occur externally as well and

are not in danger of perturbing the replay. However, allowing them can cause later analysis code to observe these changes and behave inconsistently, and these deviations can be buried deep within application code called from the analysis and hence have very subtle consequences. Other approaches based on virtualization such as Introvirt [32] and VAssert [4] allow mutations to occur within the replay process, but then revert to a prior checkpoint to undo their effects before continuing. This also prevents the analysis from maintaining any state between triggers, however, such as the depth counter in the tracing example of Section 2.5 or the lock ordering data structure in Figure 3.3, and hence restricts its power.

The same static analysis that addresses the dual address space issue can also be used to detect and reject attempts to write to the program's memory space, which is the most common way analysis code could affect the program's execution; see Section 4.7 for a more general and precise discussion of this issue. A large percentage of useful application code, however, will include side-effects such as caching even if their primary use is to read state, which would seem to imply they cannot be used in retroactive aspects.

One solution is to use `adviceexecution()`, a generic pointcut introduced in AspectJ which matches all joinpoints inside advice code. This is often used to exclude aspects from applying to other aspects when the interaction is undesirable. Combining it with the `cflow` pointcut operator creates a pointcut that covers all advice execution. In the case of retroactive aspects, it therefore covers all code in the decoupled analysis, and so can be used to only suppress or redirect side-effects in application code when called retroactively. This achieves the goal of advice code that behaves identically whether woven directly or retroactively. See Figure 3.4 for an example. Here around advice is used to avoid undesired side effects by replacing certain calls with no-ops.

Note that this simple definition will not have the desired effect if the original program also contains advice execution. However, this can easily be replaced with a more nuanced pointcut that picks out only the retroactive execution, possibly just by adding additional conjunctive clauses that name the aspects being woven retroactively.

```
1  // Program code
2  void fetchFoo(char * foo) {
3      increaseFooRefCount();
4      computeFoo(foo);
5  }
6
7  // Analysis advice
8  after() : call(bar()) {
9      char foo[100];
10     fetchFoo(foo);
11     printf("Foo is: %s", foo);
12 }
13 around() : cflow(adviceexecution())
14             && call(increaseFooRefCount()) {}
```

Figure 3.4: An example of supressing unwanted side-effects using around advice

3.5 Exploratory Implementation

This section defines our retroactive weaving prototype for ACC. The system consists of a compiler that compiles ACC aspect files into a C library and two different runtimes for evaluating the compiled aspects against a particular program execution recording. In combination with the existing ACC weaver, this allows the same ACC code to be evaluated either inline or post hoc.

3.5.1 Compiler

Our compilation process consists of a chain of several distinct source transformation phases taking ACC code as input and producing C code as output, followed by a call to an underlying C compiler (`gcc` in our case) to produce object files. The source transformation phases are implemented by extending the C Intermediate Language (CIL) OCaml library [47], which is designed to support the analysis and transformation of C source.

We have modified the CIL distribution to support parsing ACC code and representing pointcuts and advice in its abstract syntax tree. The compiler also annotates variables and types to track which memory space they refer to by leveraging CIL's support for `gcc` attributes, which are declarations with the syntax "`$attribute`"

that can be attached to nearly all elements of C syntax. In our case these annotations are either resolved `$base` or `$aspect` annotations, or annotation variables such as `$$a` that represent unknowns. Since the C type system includes value types with multiple layers of indirection (e.g. pointers to pointers), the annotations are attached to all levels of types; as an example, the type `char $base * $base * $aspect` is interpreted to mean "pointer in the aspect space to a location storing a pointer in the base program space containing an immutable character derived from program values." This would be exactly the correct type to ascribe to a variable holding the beginning of a sequence of strings extracted from the original execution: the array itself would be a consecutive region of space in the aspect execution, but the pointer values inside would refer to the memory of the original execution.

The source transformation phases are outlined below.

Annotation Inference

This phase walks the ACC source tree in a bottom-up pattern, inserting annotations and variables as needed. The `$base` and `$aspect` annotations are first introduced as base cases into the Abstract Syntax Tree (AST) according to the semantics of weaving: values that are bound from the original execution through pointcut parameters (e.g. `args(a, b, c)` and `this->args`) are assigned to the `$base` space at all levels of indirection, whereas values that are allocated by the weaving runtime (i.e. `targetName(x)` and `this->targetName` are similarly assigned to the `$aspect` space.

This phase also builds up a list of type constraints of the form `T1 = T2` based on nodes in the AST that require two address spaces to be the same: assignments, function arguments and return values, arithmetic, etc. Annotation variables are then resolved by solving the constraints using straightforward unification [58].

In general, values from different address spaces cannot be used together. This means that even arithmetic combining such values is forbidden, although explicit casts can always be inserted if necessary. An important exception is pointer arithmetic, where offset values can be from any address space, and the base pointer determines the address space of the result. This is necessary to support reading

values from arrays in aspect code, for example. Adding an annotation to represent values from either address space would allow more valid aspects to be typed without additional source modification, at the cost of added implementation complexity.

Annotation Propagation

As an additional aid to determining the address spaces of values in the source, this phase takes advantage of the observation that addresses in the original program address space cannot point to values derived from the aspect space by construction; to arrive in such a state requires an assignment to a target lvalue, which is prohibited as an illegal side-effect. Therefore, the occurrence of any type variable as an address space annotation in a type at a deeper level of indirection than a `$base` annotation can be replaced by `$base` as well. For example, `char $$a * $base * $aspect` will become `char $base * $base * $aspect`.

This is an important rule for C code, and kernel code in particular, in which it is common to calculate addresses through raw numeric calculations followed by a cast to a pointer. Ideally this rule could be incorporated into the general inferencing phase, but this rule cannot be encoded as a simple equality of types and hence cannot be included in a sound way. As future work, we plan to use a less simplistic inferencing algorithm in order to address this lack of power.

Program State Access

If the previous phases have not rejected the input program, the AST now contains only language constructs whose interactions with the program state are fully tractable. The next phase then replaces all instructions that read the program's memory with calls to reconstruct the values from the retroactive weaving runtime. This includes the reads of any lvalue derived from the program execution, or taking the address of any such lvalue. Addresses of symbols and data structure offsets are calculated from information extracted from a debug build of the program. The functions used to produce addresses of global variables and resolve register and memory accesses are declared in a header file added to the source at this stage, to be implemented by the particular backend weaving runtime.

Some special cases are implemented directly in the compiler backend since C is not expressive enough to redefine code using around advice as described earlier. For example, `printf` takes a variable number of arguments, and the operations it performs on those arguments depend on the format string; a pointer value matched with a `%s` pattern will be dereferenced, but one matched with a `%p` will not. Our solution is to match the formats to the arguments and to copy values like strings into aspect space if `printf` will dereference them, and then pass the modified arguments to the original implementation. This supports calls that pass a mix of program and analysis pointers, which is useful given that some joinpoint data consists of analysis pointers (e.g. function names as `char *` values).

Transformation from ACC to C

The final component is responsible for splitting the aspect bodies from their associated pointcuts. For a single given retroactive aspect, this component produces a C source file with three separate artifacts:

1. A set of advice bodies transformed into regular functions by discarding their pointcuts and advice kind, along with stub methods for invoking the advice body functions with a unified interface;
2. A function table for these stub methods; and
3. A string constant containing the set of aspects from the input in ACC syntax, with their bodies discarded, in the same order as in the function table.

This enables parsing the aspect stubs and dynamically invoking their body functions without recompilation of the target weaving runtime.

3.5.2 Runtime

This section describes the interface between the aspect runtime and the target execution environment and a brief description of the two target environments we currently support: one instantiates state on demand from an instruction level trace produced by the Tralfamadore [37] dynamic analysis framework, and one provides hooks into the Quick Emulator (QEMU) virtual machine monitor [11], which

we have modified to perform deterministic recording and extract virtual machine state during replay. For these backends, the retroactive aspect compiler produces a shared library containing the aspects in the original ACC source as regular C functions, with metadata consisting of the advice kind and pointcuts attached. This shared object runtime exports a common retroactive weaving interface, which serves two purposes: first, to provide the target environment with event notification callbacks on events such as function calls and returns and context switches to produce join points, and second, to let it register functions for inspecting target register and memory state.

Understanding Kernel Execution

Running aspects against kernel execution is made somewhat more complicated by context-sensitive pointcuts such as `cflow(pc)`, which matches any join point that occurs inside, or beneath, the pointcut `pc`. User-level aspects can refer to the threading abstraction provided by the operating system to uniquely identify individual flows of execution (i.e. `pthread_self()`) and track joinpoint stacks correctly.

Kernel code is more challenging because although the notion of thread exists, it is not a good abstraction for tracking individual flows of execution due to interrupts and exceptions. Interrupt handlers execute in the context of the currently running thread but are conceptually different flows of execution. Identifiers such as `current`, which maps to the currently executing task, cannot be used to distinguish between threaded (system call, kernel threads) and interrupt handler execution. Interrupts can also be nested, making this problem worse.

Because our trace-based framework is designed to analyze kernel execution, it provides an abstraction to demultiplex individual flows of kernel execution. It uses platform-specific rules to isolate system calls, interrupts, exceptions and kernel threads and label them with a unique opaque identifier. These rules are both hardware and operating system specific and require tracking stack switching, interrupts, exceptions and instruction such as `iret` and `sysexit`.

We have extended the implicit structure encoding the current join point to support the expression `this->cflowID`. This evaluates to the flow identifier pro-

vided by the backend runtime that can be used to index per-control-flow information for later retrieval, while still encapsulating the details of how independent control flows are tracked. We believe this to be a logical, generic extension to AOP and suggest that it could be used in any pointcut and advice language to avoid dependencies on specific threading libraries.

Trace-based Runtime

The Tralfamadore backend is completely offline in that running aspects involves no re-execution of guest code at all. All updates to register and memory that occurred during execution recording are present in the trace. Because Tralfamadore uses a modified version of QEMU to capture traces, it can be used to record the execution of an unmodified operating system kernel.

To add support for retroactive advice execution, we implemented a new top level operator stacked on top of the existing kernel flow and function call operators. This new operator implements a driver loop that pulls on these events and calls into the weaving runtime whenever an event of interest is recognized. Tralfamadore also supports tracking the state of registers and provides a memory index which supports efficiently finding the last update to a given memory address range. We use both of these features to support callbacks from the weaving runtime to inspect guest state.

Deterministic Replay Runtime

We have also implemented a retroactive weaving runtime directly into a virtual machine monitor (QEMU), which we have enhanced to perform deterministic execution recording and replay [25]. It records all non-deterministic events (such as external interrupts or reads of the CPU timestamp counter) occurring during execution so that they may be injected at the same point in execution during replay.

We have also added hooks into QEMU to register callbacks that can be invoked before and after the execution of any basic block. We use these hooks to track individual flows of kernel execution similarly to Tralfamadore and to track function calls and return. Whenever such an event occurs, these hooks call into the retroactive weaving library, potentially calling aspect code. Accessing guest

state is much more efficient than in the trace-based approach: it is simply a matter of mapping pointers into QEMU's data structure representing the memory of the target virtual machine. Compiled aspects and the retroactive weaving runtime are loaded and unloaded into QEMU during replay using the standard dynamic loaded library mechanism.

Deterministic recording and replay can be made very efficient through the use of CPU performance counters [25], costing as little as 5% overhead in VMWare [63]. Our prototype is much more expensive (approximately 20x) due to its pure software implementation, which makes it easier to hook instrumentation into replayed execution. As AfterSight [19] demonstrated, it is possible to use low-overhead hardware deterministic recording as the source for software replay, and we hope to do this ourselves in future work.

3.5.3 Lessons Learned

Although we we successfully authored and executed several retroactive ACC aspects using this prototype, we found that the need to provide explicit type annotations when the type inference could not determine them was a substantial burden. The second implementation described in Chapter 5 instead uses the equivalent of runtime type dispatch to address the dual address space problem, including dynamic type checks to detect incorrect usage. The overhead of this approach was not found to be significant compared to the overhead of retroactive execution in general.

In addition, we discovered that providing a language runtime that executes code using recorded state was a substantial undertaking, made especially challenging by the lack of a managed environment and adequate abstraction. Raw pointer manipulation often prevented retroactive execution from completing successfully because it was not possible to determine which address space was being dereferenced.

Finally, the presence of inline assembly in much of the Linux kernel greatly increased the engineering effort that would be necessary to support enough of the target code base to evaluate larger and more useful examples of retroactive aspects. Although time-travel programming as a paradigm is theoretically applicable to any programming language, applying it to assembly would likely face more severe

issues with easily ensuring isolation and avoiding side-effects.

3.6 Summary

We have presented the concept of retroactive advice as a unified, source-level approach to post hoc dynamic analysis. We have also described our prototype implementation of a retroactive weaver for ACC. We assert that evaluating aspects on prior executions of a program opens up a wide range of analysis applications that might otherwise be infeasible.

Chapter 4

Essential Retroactive Weaving

4.1 Introduction

Our experience with retroactive weaving motivates us to express the concept in abstract, simpler, and more precise terms. Our contribution in this chapter is a more essential and general expression of the idea, which should be applicable to any aspect-oriented language with varying effectiveness. Our presentation centres around a simple functional language called Retroactive Aspect Programming Language (RAPL), which serves as a minimally-defined example to illustrate how retroactive weaving interacts with core programming language features. We present the language and its definitional interpreter in three layers: we first present the base functional language, then extend it to support aspects, and then introduce support for retroactive weaving. Finally, we explore a general notion of soundness for retroactive weaving and explain how our example language and interpreter realize it.

4.2 Base Language

We first describe the functional core of RAPL, which we intend to reflect the essence of many modern programming language features as simply and unsurprisingly as possible. RAPL has integers, Booleans, and atomic symbols as primitive datatypes. It provides symbols for marking join points and expressing pointcuts

$$v \in \text{IDENT}, \quad n \in \mathbb{Z}, \quad s \in \text{STRING}, \quad t \in \text{TERM}$$

$$\begin{array}{rcl} t & ::= & \mathsf{true} \mid \mathsf{false} \mid (\mathsf{equal?}\ t\ t) \mid (\mathsf{if}\ t\ t\ t) \\ & \mid & n \mid (+\ t\ t) \mid (\times\ t\ t) \\ & \mid & v \mid (\mathsf{lambda}\ (v*)\ t) \mid (t\ t*) \\ & \mid & (\mathsf{rec}\ t) \mid (\mathsf{let}\ ([v\ t])\ t) \mid 'v \\ & \mid & (\mathsf{box}\ t) \mid (\mathsf{unbox}\ t) \mid (\mathsf{set\text{-}box!}\ t\ t) \\ & \mid & (\mathsf{seq}\ t\ t) \mid (\mathsf{void}) \mid (\mathsf{read}\ s) \mid (\mathsf{write}\ s\ t) \end{array}$$

Figure 4.1: Grammar for terms in the base version of RAPL

```
1 (rec (lambda (fact)
2   (lambda (x)
3     (if (equal? x 0)
4         1
5         (* x (fact (+ x -1)))))))))
```

Figure 4.2: Factorial function in RAPL

(see Section 4.3). It also includes a void value, which is returned from effectful operations. It supports first-class functions and mutable boxes, using a call-by-value evaluation strategy. Finally, it includes two operations named `read` and `write` for external input and output of integers, respectively, as these interact non-trivially with retroactive weaving. Figure 4.1 contains the RAPL expression grammar; our reference interpreter, which is in the PLAI Scheme dialect of Krishnamurthi [36], contains an equivalent `ExprC` datatype. The complete interpreter implementation is included as Section A.3.

We use the canonical factorial function as a running example; its implementation in RAPL is shown in Figure 4.2. The `rec` operator is used to create a recursive unary function.

Because retroactive weaving interacts significantly with external side-effects, we also need an example that interacts with its external environment, and hence we define a program that repeatedly prompts for a number in Figure 4.1. Again applying the `rec` operator creates a recursive function that takes a single argument, but in this case the argument is ignored and the result is a recursive thunk. If the input is zero, the program terminates. Otherwise it displays the result of applying the

Listing 4.1: RAPL code for an interactive factorial loop

```
(rec (lambda (fact_prompt)
  (lambda (v)
    (let ([in (read "x")])
      (if (equal? in 0)
          in
          (seq (write "fact(x)" (fact in))
               (fact_prompt (void)))))))))
```

factorial function from Figure 4.2 (bound to the identifier `fact` here) and prompts for the next number.

Here is an example of interacting with this program.

```
x> 3
fact(x): 6
x> 4
fact(x): 24
x> 0
Program result: 0
```

Our reference interpreter also uses the following internal definitions, which are referenced in later excerpts:

```
(define-type Value
  [numV (n number?)]
  [boolV (b boolean?)]
  [closV (arg symbol?) (body ExprC?) (env Env?)]
  [boxV (l Location?)]
  [symbolV (s symbol?)])

(define-type Binding
  [bind (name symbol?) (value Value?)])
(define Env? (listof Binding?))
(define mt-env empty)

(define Location? number?)
(define-type Storage
  [cell (location Location?) (val Value?)])
(define Store? (listof Storage?))
(define mt-store empty)

(define-type Result
  [v*s (v Value?) (s Store?)])
```

```
; Top-level evaluation function
; ExprC -> Value
(define (interp-exp expr)
  (v*s-v (interp expr mt-env mt-store)))

; Recursive interpretation function
; ExprC Env Store -> Result
(define (interp expr env sto)
  (type-case ExprC expr
    ...))
```

4.3 Adding Aspects

We now extend RAPL to include a single aspect-oriented feature: a simple form of pointcuts and advice for function application. AspectJ [34] provides the quintessential example of pointcuts and advice, in which pointcuts quantify sets of join points to affect and advice methods define how the join points are modified. Advice may be declared to run before, after, or around (i.e. in place of) a quantified join point; the latter flavour is the most expressive, and supports a distinguished `proceed` expression that represents resuming the original join point, or, in the case of overlapping advice, the next advice method in the chain.

In RAPL, the only type of join point supported is the application of functions. Since RAPL has first-class function values, around-style advice can be expressed as higher-order functions that accept and return functions. Within the body of such an advice function, invoking the passed-in function is analogous to the `proceed` expression in AspectJ, as shown in previous work [26].

Adding aspects to the base version of RAPL requires two new expression cases:

$$\begin{array}{rrcl} t \in \textsc{Term}, & t & ::= & \ldots \\ & & | & (\mathsf{tag}\ t\ t) \\ & & | & (\mathsf{aroundapps}\ t\ t) \end{array}$$

We first add a mechanism for tagging values with arbitrary metadata to the language: the expression `(tag t e)` dynamically attaches the value of the `t` expression to the value of the `e` expression. Tagged values behave identically to untagged values, except that computation involving tagged values can be identified and modified by advice. The tagging construct provides a means of identifying

join points, since otherwise function definitions have no external identity. The sub-expressions that are explicitly tagged in an expression represent the modular interface that it exports as subject to advice. In practice the associated tags would likely be derived from higher-level language features.

RAPL programs dynamically register advice using expressions of the form `(aroundapps a e)`: while evaluating the expression `e`, the advice `a` is used to potentially wrap every tagged abstraction value before it is applied to an argument. Advice takes the form of a function that accepts two arguments: the tag attached to the function being applied and the untagged form of that function. The result of the advice function is then applied to the argument in place of the original.

Here is an example of an aspect that applies to the factorial function as defined above. This aspect traces the argument and result of each recursive call to the factorial function:

Listing 4.2: RAPL code for an aspect to trace the factorial function

```
(lambda (thunk)
  (aroundapps
    (lambda (t original)
      (if (equal? t 'fact)
          (lambda (y)
            (seq (write "y" y)
                 (let ([result (original y)])
                   (seq (write "result" result)
                        result))))
          original))
    (thunk))
```

Since RAPL does not have modules, we encapsulate this aspect as a function that takes a computation which is frozen in a thunk and advises the result of activating it. Common higher-level language features such as top-level definitions and global namespaces would make composing these modules less awkward.

We tag the factorial function so the advice will apply:

```
(rec (lambda (fact)
  (tag 'fact
    (lambda (x)
      (if (equal? x 0)
          1
          (* x (fact (+ x -1)))))))))
```

We may then combine the modules by applying the aspect function to the factorial prompt thunk:

Listing 4.3: Possible input and output for applying Listing 4.2 to Listing 4.1

```
x> 3
y: 0
result: 1
y: 1
result: 1
y: 2
result: 2
y: 3
result: 6
fact(x): 6
x> 0
Program result: 0
```

Both advice and tagging may be nested. The next section provides a concrete implementation of aspect weaving that precisely defines the semantics for both forms of nesting.

4.4 Aspect Weaving

We now modify the interpreter to implement the semantics of aspects via dynamic weaving; coordinating the cross-cutting concerns as expressions are evaluated. The scope of an `aroundapps` declaration is the dynamic extent of its expression argument and hence must be tracked dynamically, rather than bound to closures as the environment is. The stack of active advice is therefore maintained in a separate argument to the interpretation function, and is generally passed down through recursive invocations of `interp`. Other forms of advice, such as advising operations on boxes, would be defined as additional type cases in the `Advice` datatype. The implementation of `tag` and `aroundapps`, represented by the ExprC cases `tagC (tag v)` and `aroundappsC (advice extent)` respectively, is trivial:

```
(define-type Advice
  [aroundappsA (advice Value?)])
(define AdvStack? (listof Advice?))

; ExprC Env AdvStack Store -> Result
```

```
(define (interp expr env adv sto)
  (type-case ExprC expr
    ...
    [tagC (tag v)
          (interp-tag tag v env adv sto)]

    [aroundappsC (advice extent)
                 (interp-aroundapps advice extent
                                    env adv sto)]
    ...))

; ExprC ExprC Env AdvStack Store -> Result
(define (interp-tag tag v env adv sto)
  (type-case Result (interp tag env adv sto)
    [v*s (v-tag s-tag)
         (type-case Result (interp v env adv s-tag)
           [v*s (v-v s-v)
                (v*s (taggedV v-tag v-v) s-v)])]))

; ExprC ExprC Env AdvStack Store -> Result
(define (interp-aroundapps advice extent env adv sto)
  (type-case Result (interp advice env adv sto)
     [v*s (v-a s-a)
          (let ([new-adv (cons (aroundappsA v-a) adv)])
            (interp extent env new-adv s-a))]))
```

The only other case directly affected is applications. The base implementation first reduces the function and the arguments to values, then invokes the `apply` routine to evaluate the function body with the augmented environment:

```
; Value (listof Value) AdvStack Store -> Result
(define (apply f args adv sto)
  (type-case Value f
    [closV (params body env)
           (let ([bs (map bind params args)])
             (interp body (append bs env) adv sto))]
    [else (error "only functions can be applied")]))
```

To handle advice, we replace this with a new version that first applies all advice in scope to the function before applying the result to the given arguments using the original `apply` routine. The core new operation is applying a single advice definition to a function, implemented by `weave-advice`. Because the actual identification of join points occurs in the source language, the interpreter implementation is relatively simple:

```
; Value (listof Value) AdvStack Store -> Result
(define (apply-with-weaving f args adv sto)
  (type-case Result (weave adv f sto)
      (v*s (v-w s-w)
           (apply v-w arg adv s-w)))))

; Applies all advice in scope for all tags on f
; AdvStack Value Store -> Result
(define (weave adv f sto)
  (type-case Value f
    [taggedV (tag tagged)
             (type-case Result (weave adv tagged sto)
               [v*s (v-w s-w)
                    (weave-for-tag adv tag v-w s-w)])]
    [else (v*s f sto)]))

; Applies all advice in scope for a single tag on f
; AdvStack Value Value Store -> Result
(define (weave-for-tag adv tag f sto)
  (if (empty? adv)
      (v*s f sto)
      (type-case Result (weave-advice adv tag
                                      (first adv) f sto)
        [v*s (v-w s-w)
             (weave-for-tag (rest adv) tag v-w s-w)])))

; Apply a single advice function to f
; AdvStack Value Advice Value Store -> Result
(define (weave-advice adv tag advice f sto)
  (type-case Advice advice
    [aroundappsA (g)
      (apply g (list tag f) adv sto)]))
```

Nested advice declarations and tags are handled with a double iteration: first over nested tags from innermost to outermost, and for each tag over nested advice declarations from inner to outer. Any tags attached to values do not affect their semantics outside of advice; for every other operation in the language the tags on each operand are stripped before performing the original logic.

4.5 Defining Retroactive Aspects

To define the semantics of retroactive aspects precisely we relate retroactive weaving, which is the process of executing them, to execution in the presence of con-

ventional weaving for AOP languages in general. We use mathematical notation to express the key requirement.

Let E be the partial function that represents evaluating a program to produce observable results, which is undefined for evaluations that do not terminate. For impure languages with external input/output, E depends not only on a term but its context, which we denote with $c \in \text{CTXT}$. The range of E is OBS; for RAPL this is the result value and any output. Therefore $E : \text{PGM} \times \text{CTXT} \to \text{OBS}$.

As outlined in Section 2.4, Masuhara and Kiczales [43] model AOP in general via a weaving process with signature $W : \text{A} \times \text{B} \to \text{X}$. Retroactive weaving applies to those instances of AOP frameworks where at least one of the input languages is executable independently and its semantics subsumed by X. This is true of AspectJ, for example, but not of DemeterJ [39]. WLOG, we assume the executable language is B, and relabel the weaving signature as $W : \text{ASPECT} \times \text{PGM} \to \text{PGM}$. Therefore executing an aspect a together with a program p in context c is modelled as $E(W(a,p),c)$.

We augment E to also produce a trace, an abstract notion of intermediate computation information: $E_{traced} : \text{PGM} \times \text{CTXT} \to \text{OBS} \times \text{TRACE}$.

Retroactive weaving is another evaluation function $RW : \text{ASPECT} \times \text{TRACE} \to \text{OBS}$. Its behaviour is defined by an invariant: for any program p, aspect a and execution context c, if $E_{traced}(p,c) = (o,t)$, and $RW(a,t)$ is defined, then $RW(a,t) = E(W(a,p),c)$.

The intuition behind this model is that a trace encodes partial information about the past execution context. Retroactive weaving produces the observable behaviour the augmented program would have produced in that context. If the augmented program depends on missing information, the process must signal an inconsistency error, represented here by undefinedness in RW.

4.6 Retroactive Weaving

We now describe how our definitional interpreter provides both tracing and retroactive weaving.

4.6.1 Recording and Reading Traces

The augmented interpreter must be able to record the relevant information during one execution and read this information during a future execution. Thus we first require interpretation to record a subset of the states it reaches while interpreting externally, so that retroactive weaving can identify join points post-hoc and apply advice as required. Within this context we use *trace* to refer to an ordered list of recorded interpretation states.

To consume traces one state at a time during retroactive execution, we add another parameter to the interpretation function for the remaining trace to read. The head of this trace represents the current state of the original execution; under normal evaluation this list will be empty. The retroactive weaving process reacts to each recorded state, potentially performing additional execution, before moving to the next recorded state by popping the head of the trace list.

```
; ExprC Env AdvStack Store Trace -> Result
(define (interp expr env adv sto tin)
  ...)

(define Trace? (listof State?))
(define mt-trace empty)

; Trace -> State
(define (trace-state tin)
  (first tin))

; Trace -> Trace
(define (next-trace-state tin)
  (rest tin))
```

To produce traces, we extend the `Result` data type so that every computation can also provide the trace for that computation. In addition, during retroactive weaving the input trace must be threaded through the interpreter much as the store is, and hence we also extend `Result` to include the remaining input trace:

```
(define-type Result
  [v*s*t*t (v Value?) (s Store?)
           (tin Trace?) (tout Trace?)])
```

The datatypes that define our version of tracing are as follows:

```
(define-type Control
  [app-call (f Value?) (args (listof Value?))]
```

```
  [app-result (r Value?)])
(define-type State
  [state (c Control?) (adv AdvEnv?) (sto Store?)
         (tin Trace?)])
```

The `Control` datatype enumerates the different kinds of interpretation control flow we record, in particular just before applying a function value to an argument value, and just after such a call produces a result value. The `State` datatype combines the interpreter control point with the interpreter arguments. It does not include the environment because advice in RAPL is modelled with function calls, and hence cannot access the lexical scope of join points, but this would not necessarily be true in AOP languages that make use of dynamic binding in aspects.

Since our traces only carry information about function applications, only the main application routine extends the current trace:

```
; Value (listof Value) AdvEnv Store Trace? -> Result
(define (apply-with-weaving f args adv sto tin)
  (type-case Result (weave adv f sto)
    (v*s*t*t (v-w s-w tin-w tout-w)
      (type-case Result (apply v-w args adv s-w tin-w)
        (v*s*t*t (v-r s-r tin-r tout-r)
          (let ([c (state (app-call f args)
                          adv sto tin)]
                [r (state (app-result v-r)
                          adv s-r tin-r)])
            (v*s*t*t v-r s-r tin-r
                     (append (list c) tout-w tout-r
                             (list r)))))))))
```

Other operations in RAPL could be recorded in the same style; in general these traces are produced by appending the sub-traces for individual sub-computations together in evaluation order.

We omit the details of serializing these traces to and from persistent storage, in our case one file per trace, as they do not affect the semantics of retroactive weaving. In practice, however, there is ample opportunity for optimizing the writing and reading of such traces, and scalable implementations can be quite sophisticated [55]. In particular, recording a full copy of the store at every event can be expensive, and more practical implementations will instead record individual changes to the store incrementally.

In all but the simplest of programming languages and their environments, it will

not be possible or feasible to record all of the information a program could have queried during its execution. This is especially true of more mainstream languages that have access to file systems, networks, and more unpredictable sources of values such as the current time. The particular instance of tracing we present here is chosen to be simple and sufficient to support a reasonable number of retroactive aspects, and we intentionally omit many other interpreter states as well as the external input accessed via `read` during execution. For a more complete discussion of how this affects our implementation's completeness, see Section 4.7<>.

4.6.2 Retroactive State

The state of the retroactive interpretation can build on the original interpretation state. In particular, retroactive execution can use references to values, including store locations and their contents, from the original execution. Consider an alternative version of the factorial function which uses a box internally to track its counter:

```
(let ([fact_helper
       (rec (lambda (fact)
         (tag 'fact_helper
              (lambda (bx)
                (let ([x (unbox bx)])
                  (if (equal? x 0)
                      1
                      (seq (set-box! bx (+ x -1))
                           (* x (fact bx)))))))))])
  (lambda (x) (fact_helper (box x))))
```

To create an equivalent version of Figure 4.2 for this implementation, the advice needs to dereference the box passed to the helper function in order to obtain the actual value of `x`. Therefore, the retroactive weaving interpretation must deal with a mix of locations: new locations created during the retroactive evaluation and old locations from the trace. In addition, the values stored in old locations may change as the trace is traversed, so references to old locations must somehow be kept current. Finally, it is necessary to distinguish old and new locations to detect inconsistent executions (see Section 4.7).

Our approach is to add another case to the `Value` datatype to implement a layer of indirection on values obtained from the trace. This aligns closely with

how production implementations are likely to be implemented, as it allows the underlying trace and its store to proceed independently of the retroactive state [60].

```
(define-type Value
  ...
  [traceValueV (v Value?)])
```

When a value from prior state is bound by a retroactive aspect (such as the box passed to advice for the `fact_helper` function above), it must be lifted to the retroactive context, so that new and old store locations can be distinguished. The value may be a box itself, or it may be a compound value such as a closure which may transitively refer to store locations. We define a `lift-trace-value` function in our interpreter for this purpose. The omitted `Value` cases are handled by straightforward structural recursion: primitive values are untouched, and tagged values and the value bound by the environments stored in closures are lifted piecewise.

```
; Value -> Value
(define (lift-trace-value v)
  (type-case Value v
    [boxV (trace-loc)
          (traceValueV v)]
    ...))
```

We then augment fetching from the store to handle boxes from the trace:

```
; Store Trace Value -> Value
(define (fetch sto tin b)
  (type-case Value b
    [boxV (loc) ...] ; As before
    [traceValueV (v)
                 (type-case State (trace-state tin)
                   [state (c adv s-t tin-t)
                          (fetch s-t tin-t v)])]
    [else (error 'interp "attempt to unbox a non-box")]))
```

4.6.3 Retroactive Control

Implementing retroactive weaving involves producing the extra execution that an aspect specifies at various positions in the trace. When an application callback is applied retroactively to an application in the trace, we need to use a placeholder

to resume the original execution - that is, reading the rest of the trace - instead of evaluating the application. To achieve this we add another case for values:

```
(define-type Value
  ...
  [resumeV])
```

Any tags on the original function must be carried over to the stub value so that application advice will behave identically:

```
; Value -> (listof Value)
(define (all-tags v)
  (type-case Value v
    [taggedV (tag tagged)
             (cons tag (all-tags tagged))]
    [else empty]))

; (listof Value) Value -> Value
(define (deep-tag tags v)
  (foldr taggedV v tags))

; Value -> Value
(define (rw-resume-value v)
  (deep-tag (all-tags v) (resumeV)))
```

Applying this value as if it were a function instead resumes the process of weaving the trace:

```
; Value (listof Value) AdvEnv Store Trace -> Result
(define (apply-without-weaving f args adv sto tin)
  (type-case Value f
    [closV (params body env)
           (let ([bs (map bind params args)])
             (interp body (append bs env) adv sto tin))]
    [resumeV ()
             (rw-call f args adv sto tin)]
    [else (error "only abstractions can be applied")]))
```

The core of the retroactive weaving implementation are these three mutually recursive functions:

```
; Value (list of Value) AdvEnv Store Trace -> Result
(define (rw-call f args adv sto tin)
  (rw-result adv sto (next-trace-state tin)))

; Value (listof Value) AdvStack Store Trace -> Result
(define (rw-replay-call f args adv sto tin)
```

```
  (let ([resume (rw-resume-value f)]
        [lifted-args (map lift-trace-value args)])
    (apply-with-weaving resume lifted-args adv sto tin)))

; AdvStack Store Trace -> Result
(define (rw-result adv sto tin)
  (type-case State (state-c (trace-state tin))
    [app-call (f args)
      (type-case Result (rw-replay-call f args
                                        adv sto tin)
        (v*s*t*t (v-r s-r tin-r tout-r)
                 (rw-result adv s-r
                            (next-trace-state tin-r))))]
    [app-result (r)
      (v*s*t*t (lift-trace-value r)
               sto tin mt-trace)]))
```

`rw-replay-call` consumes the next sub-sequence of the trace from a function application up to its corresponding result, and `rw-result` continues to consume such sub-sequences until it reaches the result for the current application. These routines essentially reconstruct the original tree of recursive calls to the interpretation function. Note that these versions of the core retroactive weaving routines do not produce a trace for retroactive weaving itself in order to simplify presentation, but adding this tracing using the implementation strategy shown in Section 4.6.1 is straightforward.

The top-level entry point to retroactive weaving is `interp-rw`, a separate but related function that corresponds to *RW* in the abstract model in Section 4.5. As demonstrated above, in RAPL advice can be declared in discrete modules if the computation they advise is provided as a parameter, delayed within a thunk. The `resumeV` value is also used to represent the trace as such a thunk.

4.7 Ensuring Soundness

The implementation so far will behave correctly for many retroactive aspects. However, not all retroactive aspects are sound according to the semantics defined in Section 4.5; if the augmented program attempts to access information that was not recorded, retroactive weaving is required to terminate in an error, whereas the interpreter thus far may instead produce inconsistent observable behaviour.

The implementation above assumes that when a recorded state has been processed and the paused original execution resumed, that original execution would have reached the same next recorded state in the trace. If a retroactive aspect perturbs the program state in some way, the program may have continued to make an unsupported operation as above, so we cannot assume this is safe. Therefore, for this particular implementation of tracing and weaving, ensuring soundness is equivalent to ensuring that retroactive advice would not have perturbed the original execution.

Since aspects in RAPL are quite general and expressive, there are several ways that retroactive weaving can fail:

New external side-effects: Advice itself might attempt to add additional interaction with the original context. In RAPL this means extra calls to `read`, which conceptually consume values from the program input prematurely and shift the values read by the original program, leaving the later inputs uncertain. This is preventable by replacing the source drawn by the `read` expression with a stub that raises an error during retroactive weaving.

Modifying arguments: Advice may pass a different list of arguments to the wrapped function than was originally provided. To prevent this, we add within the `rw-call` function a comparison of the arguments the stub `resumeV` value is applied to against the arguments provided in the original join point.

Modifying results: Similarly, advice may return a different result than a join point of the original computation. Another check must be inserted before returning from `rw-replay-call` to compare the value produced by the advice stack to the original.

Modifying the original store: Advice could also perturb the original execution more indirectly by mutating boxes, so we modify the implementation of `set-box!` to raise an error if the given box is a reference to the original store (i.e. a `traceValueV` as described in Section 4.6.2).

Modifying control flow: More deviously, advice may fail to invoke advised functions, or invoke them more than once. Because the construct that represents proceeding in advice is a first-class value (i.e. a function), it could also be bound and applied later, outside the scope of the advice. All these cases can be prevented by attaching the length of the remaining trace to the stub `resumeV` value

at the time it is created, and comparing this to the current length of the remaining trace in the store value whenever it is applied. This ensures each stub is applied in order and no more than once. An additional check after the top-level call to `rw-replay-call` to verify that the entire trace has been consumed ensures that each stub is applied at least once.

4.7.1 Deterministic Replay

The restrictions above depend heavily on the exact information recorded during the original execution. Rather than recording the full state of interpretation at relevant points, which can be very expensive, the runtime could instead only record non-deterministic events, so that the state can be reconstructed by replaying the interpretation. For RAPL, recording would become storing only the sequence of input integers, and replaying would involve assigning the read source to be that sequence.

The straightforward approach to retroactive weaving using replay is to trace the replay process and then use retroactive weaving on the trace as above. This could be made more efficient by having the replay process produce a stream of states which the weaver consumes. It is tempting to optimize this further by directly weaving the retroactive aspects against the original program during the replay interpretation instead. This is not sound in general, though, without again modifying the interpreter to guard against new retroactive external side-effects as above.

4.8 Related Work

RAPL bears a strong resemblance to AspectScheme [26], another aspect-oriented language with first-class function values. The key place they differ is that AspectScheme is an AOP extension to an existing full-featured programming language, whereas RAPL is intended to be a core language, with the minimum features required to support retroactive weaving. Some of AspectScheme's features, such as statically scoped advice and equality of functions via source location, can be expressed via desugaring to RAPL.

De Fraine et. al. provide a core calculus for AOP with their A calculus [23]. The A calculus is object-oriented, but like RAPL also models proceed as binding a

closure-like value, and supports passing said closures as first-class values. Because the RAPL interpreter is more focused on modelling two alternative strategies of aspect weaving, it avoids object-orientation and types to keep the semantics simpler.

4.9 Summary

We have presented the concept of retroactive weaving as an abstract concept directly related to the semantics of conventional aspect weaving for aspect-oriented programming languages. We provided a definitional interpreter that implements retroactive weaving for a simple core language, illustrating the interaction of retroactive weaving with common core language features. Finally, we discussed the soundness requirement for such an implementation and its consequences.

Chapter 5

Retroactive Execution on the JVM

This chapter focusses first on the retroactive execution facet of time-travel programming, as it provides significant utility independent of retroactive weaving, and its efficient implementation for a full-featured modern language is non-trivial. We present an architecture that enables additional execution in the context of a program snapshot, as if a live, debuggable process had been restored from the snapshot. This allows developers to invoke any code present in the original process, or even to load new analysis code into the emulated process, with no need for metaprogramming. This functionality is implemented as an ordinary library and does not require a custom language runtime. The execution is made sound by forbidding the recorded objects from accessing state external to the snapshot, since the original environment has been lost and cannot be accurately emulated. We hence refer to these objects as *holographic objects*: accurate recreations that cannot interact directly with the outside world.

The abstract model used to represent the state in a program snapshot for holographic objects can also represent any instantaneous state within an execution recording over time. Chapter 6 hence expands on this architecture to implement retroactive weaving.

The main contributions of this chapter are:

- an architecture for holographic objects, which enable restricted execution starting from the state captured in a heap dump;
- evidence that this architecture can be efficiently implemented in a statically-typed language on an unmodified commodity language runtime; and
- evidence for the utility of holographic objects by using them to diagnose an unsolved memory leak in a mature mainstream application.

The implementation discussed here supports the Java programming language and runs on the JVM. The general approach based on emulating language semantics, however, is applicable to other language runtimes as well; see Section 5.1.3 for a discussion of the requirements to support holographic execution efficiently.

5.1 Holographic Virtual Machines

Because it offers high fidelity, a snapshot of a process at the time of a failure is a nearly omnipresent feature of modern programming language runtimes, and often translates into a heap dump file that contains the state of every live object and its connections to other objects. A number of tools can parse this file and present the developer with an interactive, browsable tree of values. This interface is familiar and useful for developers, as it parallels how a debugger models the state of a live system. Unlike live debugging, however, the developer cannot execute any of their own code in the context of the failure, which is a critical piece of functionality as illustrated in Section 3.1.

Ultimately all of these difficulties would be resolved if a heap dump analysis tool could execute code on the objects in the snapshot, as if the execution occurred on the live process immediately after the snapshot was taken. We aim to provide this functionality through *holographic objects*, which are virtual objects that reflect the state and behaviour of the objects recorded in the snapshot. This section describes the high-level architecture we have used to make this possible.

To implement holographic objects, we require a reflective API for accessing the state of the recorded objects, and an execution environment that implements the semantics of normal execution with respect to the reflective API instead of in-memory native objects. For example, an instruction that accesses a field of an

object should have the effect of accessing that field from the holographic object in the heap dump via our reflective API.

Holographic objects should behave like the recorded objects they imitate, or else any analysis performed on them may produce incorrect results. Any code in the control flow of holographic execution that cannot be exactly reproduced based on the information in the heap dump must result in an explicit exception. This includes any attempt to access or mutate the external environment of the original process, such as other processes, the file system, the network, and so on. This also implies that holographic objects must be completely sandboxed: it must be impossible for them to obtain references to any objects in the host Virtual Machine (VM), or vice versa. Holographic objects are hence encapsulated inside a *holographic virtual machine*. Values may only be passed between the guest and host VMs using explicit reflective methods, and only primitive values[1] are permitted to avoid leaking references.

Figure 5.1 contrasts an ordinary VM with a holographic VM running inside another ordinary VM. On the left is an ordinary virtual machine and its interactions with the file system and external environment. On the right, a holographic VM simulates the behaviour of a VM restored from a heap dump. A holographic VM is only permitted to read class files from disk and interact with the VM emulating it through reflective methods. It is otherwise forbidden from interacting with its environment.

[1] In the case of the JVM, `String` values are permitted despite being objects, as they are a core type whose implementation must be immutable.

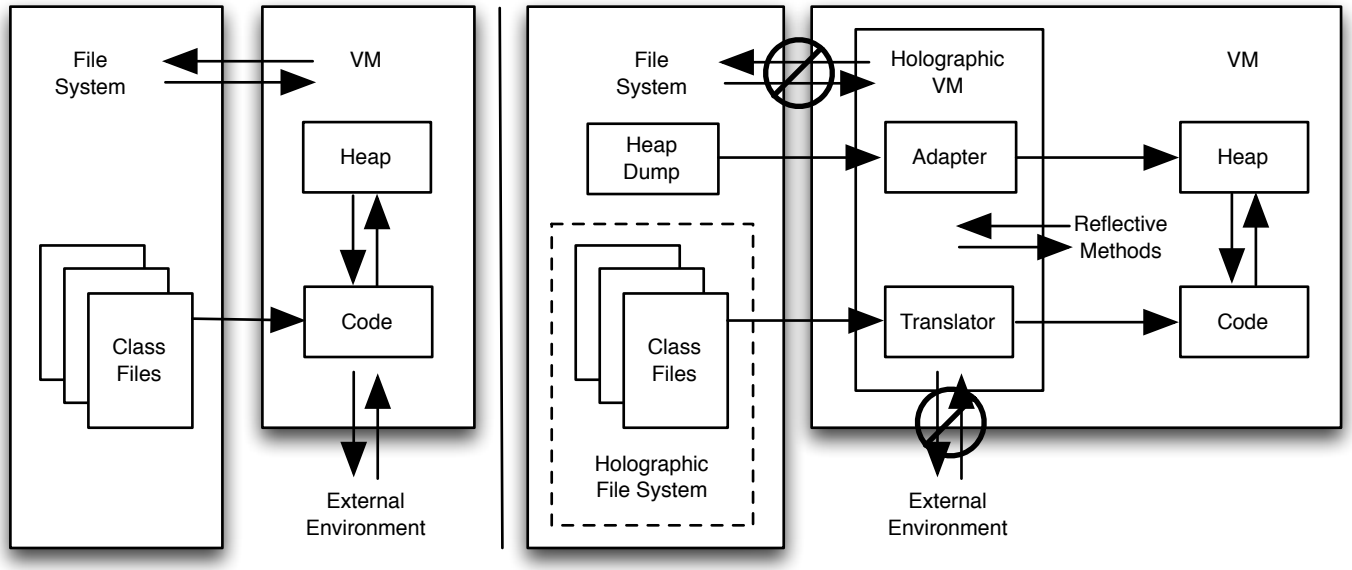

Figure 5.1: The overall holographic objects architecture

5.1.1 Mirrors

To support creating holographic objects on top of multiple snapshot formats, or indeed to other sources of object state, we define our reflective API using an independent set of reflective interfaces. The core functionality of the reflective access we need is apparent if we compare the heap dump model to existing reflective APIs. The Java platform includes two such APIs: a set of built-in reflective methods such as `Class.getFields`, and the Java Debugging Interface (JDI) provided by the Java Platform Debugger Architecture [50], on which remote Java debuggers are built. Each API provides similar functionality backed by different state: the built-in Java reflection methods reflect on the state of the current VM, the JDI reflects on the state of a separate VM being debugged, and the heap dump model reflects on the past instantaneous state of another VM.

Bracha and Ungar [13] label such pluggable, independent reflection interfaces as *mirrors*, and we adopt their terminology here. We define a central interface named `VirtualMachineMirror`, which encapsulates an entire object graph including all loaded classes and hence all executable code in the system. Other interfaces represent objects, classes, fields, methods, arrays, and so on.

A holographic VM is then represented as a `VirtualMachineHolograph` wrapper that refers to an underlying `VirtualMachineMirror` instance and implements that interface itself. This achieves our goal of making holographic objects a general-purpose library, as the wrapper can be applied to any representation of VM state that can be used to implement the generic mirrors API. The library provides similar holographic wrappers for the other related mirror interfaces. Most importantly, the holographic implementation of `MethodMirror.invoke` does not delegate to the wrapped method mirror, but instead emulates the semantics of that method's definition as described above.

5.1.2 Mutations

Although holographic execution is not allowed to read from or write to the state of the outside world, many useful expressions that are semantically functional will have internal side-effects that attempt to modify the internal object graph. For example, looking up a string key in a `HashMap` as described earlier requires cal-

culating the string's hash code, and the implementation of string hashing caches the result in an instance field of the string the first time it is calculated. This means if the method was not previously called on a string in the original VM, or if the string was newly-created as part of holographic execution, the method will attempt to set a new value on the mirror. The heap dump model is read-only, as many potential mirror implementations will be, and hence this will trigger an exception.

To support mutation in holographic execution, the holographic adapters superimpose a mutable mirror graph over the wrapped, potentially immutable graph. Each wrapper is initially empty, exposing state identical to that of the wrapped object. As values are written to the holographic objects they are stored in the wrapping object, and future reads will return those values. The same approach applies for other more subtle side-effects on the mirrors API, such as expanding the set of classes loaded by a class loader by defining a new class. This polymorphism between old and new state in the model is analogous to the polymorphism in the RAPL interpreter's `Value` type between the `traceValueV` data constructor and the remaining data constructors, as described in Section 4.6.2.

Maintaining mutations on the object graph independently in this way also offers flexibility in the semantics of multiple successive sessions of holographic execution. If the same holographic VM instance is used for each, the side-effects of prior executions will potentially affect future executions. This is consistent with a developer's experience when evaluating expressions in a normal debugging client. Alternatively, a new holographic VM can be instantiated for each evaluation and then discarded, so each successive evaluation proceeds from the same pristine initial snapshot state. This is equivalent to experimenting with a live process by repeatedly forking a new sandboxed process that can be perturbed in arbitrary ways and then discarded.

5.1.3 Translating Code

The most obvious approach to implementing holographic execution would be customizing a language runtime to support it, but this would be non-portable. We instead chose the implementation strategy of translating programs into lifted versions equivalent to holographic object semantics. Each instance of a core language

operation is mapped to methods of the mirrors API, and the implementors of those interfaces thus determine the runtime behaviour of the language.

This general approach applies to any programming language in which it is possible to express a large subset of the language's semantics in a reflective interface. Object-orientation is not a hard requirement, as illustrated by the implementation for C described in Section 3.5; in that context a collection of methods such as `byte readMemory(void * addr)` provide the equivalent interface to mirrors. In addition, holographic execution will be more performant and less complicated to implement if at least some of the operations in the language have copy-by-value semantics, since those operations can be left untranslated and hence operate at native speed. Holographic execution works well for JVM bytecode since it is not purely object-oriented: primitive values are passed by value and primitive operations are not customizable by user programs.

Listing 5.1: Original Java code for the sample Employee class

```
1  public class Employee {
2
3    private int age;
4
5    public static Set<Employee>
6      over40(Employee[] input) {
7
8      Set<Employee> result =
9        new HashSet<Employee>();
10     for (Employee e : input) {
11       if (e.age > 40) {
12         result.add(e);
13       }
14     }
15     return result;
16   }
17 }
```

Listing 5.2: Original JVM bytecode for the sample Employee class

```
1  public class Employee {
2
3    private I age
```

```
4
5    public static over40([LEmployee;)LSet;
6      L0
7       NEW HashSet
8       DUP
9   1)| INVOKESPECIAL HashSet.<init> ()V
10      ASTORE 1
11     L1
12      ALOAD 0
13      DUP
14      ASTORE 5
15  2)| ARRAYLENGTH
16      ISTORE 4
17      ICONST_0
18      ISTORE 3
19      GOTO L2
20     L3
21      ALOAD 5
22      ILOAD 3
23  3)| AALOAD
24      ASTORE 2
25     L4
26      ALOAD 2
27  4)| GETFIELD Employee.age : I
28      BIPUSH 40
29      IF_ICMPLE L5
30     L6
31      ALOAD 1
32      ALOAD 2
33  5)| INVOKEINTERFACE Set.add (LObject;)Z
34      POP
35     L5
36      IINC 3 1
37     L2
38      ILOAD 3
39      ILOAD 4
40      IF_ICMPLT L3
41     L7
42      ALOAD 1
43      ARETURN
```

```
44 }
```

Listing 5.3: Translated JVM hologram bytecode for the sample Employee class

```
1  public class hologram/Employee
2      extends ObjectHologram  {
3
4    // Inherited from ObjectHologram:
5    // public LObjectMirror; mirror
6    public final static LClassMirror; classMirror
7
8    public static over40(Lhologramarray1/Employee;)
9      Lhologram/Set;
10    L0
11     LINENUMBER 17 L0
12     NEW hologram/HashSet
13     DUP
14 1)| GETSTATIC hologram/HashSet.classMirror
15   |    : LClassMirror;
16   | INVOKEINTERFACE ClassMirror.newRawInstance
17   |    ()LInstanceMirror;
18   | INVOKESPECIAL hologram/HashSet.<init>
19   |    (LInstanceMirror;)V
20     ASTORE 1
21    L1
22     ALOAD 0
23     DUP
24     ASTORE 5
25 2)| INVOKEINTERFACE ArrayMirror.length ()I
26     ISTORE 4
27     ICONST_0
28     ISTORE 3
29     GOTO L2
30    L3
31     ALOAD 5
32     ILOAD 3
33 3)| INVOKESTATIC ObjectArrayHologram.getHologram
34   |    (LObjectArrayMirror;I)LHologram;
35   | CHECKCAST hologram/Employee
36     ASTORE 2
```

```
37      L4
38        ALOAD 2
39   4)|  GETSTATIC hologram/Employee.classMirror
40     |      : LClassMirror;
41     |  LDC "age"
42     |  INVOKESTATIC InstanceHologram.getIntField
43     |      (LHologram;LClassMirror;LString;)I
44        BIPUSH 40
45        IF_ICMPLE L5
46      L6
47        ALOAD 1
48        ALOAD 2
49   5)|  INVOKEINTERFACE hologram/Set.add
50     |      (LHologram;)Z
51        POP
52      L5
53        IINC 3 1
54      L2
55        ILOAD 3
56        ILOAD 4
57        IF_ICMPLT L3
58      L7
59        ALOAD 1
60        ARETURN
61  }
```

Our particular implementation targets the JVM, and hence the source language it translates is JVM bytecode using the ASM bytecode processing framework [14]. We label the translated versions as *hologram* classes. These classes are encapsulated within the holographic VM API and never directly exposed to developers or tools using holographic objects. All instance field declarations in the original classes are replaced with a single `ObjectMirror` instance field, and the individual bytecode instructions are lifted to operate on those instances. The transformations are all local and context free, although a single instruction will frequently be translated into multiple instructions. Figures 5.1, 5.2 and 5.3 contain a small example of how bytecode is transformed. Modified instructions are numbered in both the original and translated bytecode listings. Only object and array instructions are modified; control flow and primitive instructions are left untouched. Note

that downcasts such as the one in Figure 5.3 are often necessary to ensure type safety.

The semantics of holographic execution imply that holographic object references require two orthogonal dimensions of polymorphism: the original class hierarchy for virtual and overloaded method invocations, and the virtual mirror interface methods for object state access. We have chosen to map the original hierarchy into an isomorphic hierarchy of hologram types which preserve the subtype relation, allowing method invocation to operate as in the original bytecode. Our techniques are similar in several respects to those used by Factor et. al. [27] to transparently rename classes in order to support instrumentation of core Java classes. Note that another approach would be to replace object references with direct mirror references and implement dynamic method dispatch manually instead. We suspect that the overhead of handling method dispatch is likely equal to or worse than the overhead of wrapping mirrors with hologram class instances.

Each source type is usually mapped to exactly one internal type, but in some cases maintaining the subtyping relationship in user-level code requires splitting the type into a concrete class and an interface. The mapping function between hierarchies is therefore actually defined by two functions: HC, which is guaranteed to be a concrete, instantiable class, and HT, which may be an interface. These functions obey the following properties:

- For all types C in the original bytecode, $HC(C) <: HT(C)$
- For all types C and D in the original bytecode, if $C <: D$, then $HT(C) <: HT(D)$

In general, $HC(C)$ is used wherever new instances of C are created, or when C is used as the superclass of another class, whereas $HT(C)$ is used wherever C is used as a reference type for local variable, method parameters, and so on. The cases where the two functions differ are outlined below.

Interfaces: `Object` is both the base class of all concrete classes and the top of the subtyping lattice, and hence a supertype of interfaces as well. References of type `Object` are mapped to a `Hologram` interface, which all hologram classes and interfaces implement and extend, and which has a single `getMirror()`

method. Where Object appears as a superclass, however, an `ObjectHologram` class is used instead, which actually declares the mirror field and implements `getMirror()`.

Arrays: Each distinct array type, which is normally created automatically in the JVM without requiring explicit class definitions in source, is mapped to a distinct class type; although there is no virtual method dispatch on arrays, array types can still create valid method overrides when used as parameter types. These must also be split, since they must be concrete and instantiable but also support multiple inheritance because of covariance; for each interface `A` that `B` implements, the hologram type for `B[]` must be a subtype of the hologram type for `A[]`.

If T is an array type with reference element type E and n dimensions (i.e. the type `E[][](n)[]`), we use $HAC(E,n)$ and $HAT(E,n)$ to refer to $HC(T)$ and $HT(T)$, which will be a class and an interface, respectively. The extends and implements clauses for these types are defined according to the following rules:

- $HAC(E,n)$ `implements` $HAT(E,n)$
- If E `extends` C, then $HAT(E,n)$ `implements` $HAT(C,n)$
- If E `implements` I, then $HAT(E,n)$ `implements` $HAT(I,n)$
- For all $n > 0$, $HAT(\texttt{Object},n)$ `implements` $HAT(\texttt{Object},n-1)$

The last rule above is necessary because of array subtyping covariance and the fact that `Object[]` $<:$ `Object`. Note that $HAT(\texttt{Object},0)$ is simply $hologramType(\texttt{Object})$, which is the `Hologram` interface.

Since the results of translating bytecode will be the same for successive holographic VMs on the same heap dump, our system caches translated bytecode on disk to improve performance. The cache is a fast associative mapping keyed by class name with sequential separate chaining to handle multiple classes with the same name. This approach is effective since the name of a class is by far its most specific characteristic, but still handles multiple classes with the same name occurring in a single VM.

Holographic JVMs also provide an optional *prepare* operation that iterates through all currently loaded classes and eagerly generates the translated bytecode

for each, which will pre-populate the cache. This will often be the preferred workflow: a holographic JVM could be prepared in advance and the cached bytecode distributed along with the heap dump.

5.2 Scope

There are several obstacles that may prevent holographic execution from emulating live execution soundly. All are direct results of missing information in the snapshot, although in most cases these can be solved by additional configuration provided by the user. This section outlines the factors that limit the completeness of this technique and the extent to which we are able to overcome them in our implementation.

5.2.1 Missing Bytecode

The most immediate obstacle to holographic execution on the JVM is the fact that heap dumps generally do not contain any bytecode, as most JVM implementations maintain class definitions in a separate area of memory. We must somehow recover the definitions of the classes in the heap dump in order to execute any code. Class definition on the JVM is dynamic: the core `ClassLoader` class and its subclasses are used to locate the bytecode for a requested class name at runtime. The implementation of these class loaders can be arbitrarily complex and is often non-trivial in popular application containers such as OSGi, and so providing the missing bytecode in a holographic VM though manual configuration is not feasible.

Our solution is to leverage the fact that nearly all class loaders eventually load bytecode from a class file on the file system, and more specifically one that matches the requested class name. We use holographic execution itself to call the appropriate method on the class' loader to read the contents of the matching class file. This approach is valid for the vast majority of Java code, but for full generality this piece of the architecture is pluggable so that more unusual class loaders that dynamically modify or generate bytecode can be handled when the generic solution fails to locate a class file.

Since the state of the original file system at execution time is also not captured

in the heap dump, an exception would normally be thrown when this mechanism attempts to access the file system. However, the configuration of a holographic VM includes a simple finite mapping from paths in the original file system to paths in the file system of the host machine, creating what we call a *holographic file system*. Whenever holographic execution attempts to access the original file system, the path is remapped onto the host file system. This approach also works for paths inside compressed class file archives ("jar" files), and could theoretically be extended to support more atypical sources such as URLs.

This system, in combination with other hooks for external input and output described in the following section, resembles how the KLEE symbolic execution tool [15] simulates the external environment. Where KLEE represents values and control flow affected by the external environment symbolically, the holographic VM requires an exact simulation of the past environment and does not tolerate uncertainty, instead raising an error to indicate missing information.

Assuming that the developer using this system can provide a copy of the same version of the compiled class files, which should not be difficult given the prevalence of source version control systems in software development, the holographic file system allows any class loading logic to read the correct bytecode from disk. This solution also allows the holographic VM to load classes that were available to the original VM but not yet loaded, which is often necessary when holographic execution hits code that hadn't yet been executed in the original VM. Furthermore, another workaround for the problem of unusual class loaders above is to pre-generate the relevant bytecode as class files in the mapped file system.

5.2.2 Native Methods

Many programming language platforms feature standard runtime libraries that are impure, in that some of the provided features are not implemented in the language but instead handled by the runtime itself with no corresponding source code. In the case of the JVM, many low-level methods in the core Java Runtime Environment (JRE) library are native methods, which means they have no bytecode but are instead handled directly in the JVM. Such native methods cannot be called directly on holographic objects, but holographic execution will inevitably encounter them.

Even calling a `toString` method as described earlier is almost guaranteed to hit the native `System.arraycopy` method somewhere in its control flow.

Native methods may have arbitrary effects on the external environment of the VM, and hence some cannot be called in holographic execution. Many are purely functional in behaviour, however, and are only implemented in native code for efficiency. The holographic VM architecture includes another pluggable mechanism for providing semantically equivalent, Mirror-based Native Methods (MNMS), and includes such implementations for the most commonly encountered native methods in the JRE.

In most cases the implementations of MNMs can be quite naive and unoptimized. In the context of supporting post-hoc debugging and analysis, the raw efficiency of the implementation is not the primary concern so long as the method's semantics can be accurately reproduced. See Section 5.3.2 for a discussion of the efficiency of our architecture.

Native methods can be left unimplemented, or they can be expressly marked as forbidden because their semantics require accessing their external environment. In either case, the unsupported native method is replaced with an MNM stub that throws an exception. This means that classes with unimplemented or forbidden native methods can still be loaded and used in holographic execution so long as those native methods are not actually called. This is critical since the classes in the JRE include over 1000 native methods, many of which involve some form of input from or output to the external environment.

Note that application classes outside of the standard language runtime can also include native methods, and so if a developer wishes to execute holograph code that will hit those methods they must provide the required MNMs themselves. Native methods are much less prevalent in application code than in the core JRE, however, and the burden of providing these alternate implementations is far less than the burden of re-implementing everyday code as in the reflection-based analysis approach.

5.2.3 Class Initialization

Class initialization occurs in Java when a class is first used, and involves invoking a special static method in the class' bytecode called an *initializer*. This can have arbitrary effects on the object graph, and holographic execution must preserve this behaviour by invoking the initializer of any uninitialized class before accessing it.

Like a class' bytecode, however, the initialization flag is not present in most heap dumps, so there is no direct way to tell if a class was defined but not yet initialized at the time of the dump. Since failing to initialize an uninitialized class can lead to inconsistent, unsound errors, holographic execution must raise an exception if it attempts to load a class whose initialization status is indeterminate.

We observe that in almost all cases the initialization status of a class can be automatically inferred from other data, based on the rules for when initialization must occur. Before class initialization, every non-constant static field has a default value: `false` for boolean fields, `null` for object references, and so on. Setting a value on a static field forces initialization, so a non-constant static field in the heap dump with a non-default value implies that the class must be initialized. Conversely, if the execution of a class initializer has the definite effect of setting a non-default value on a field but that field has the default value in the heap dump, the class must not be initialized.

In addition, given the definition of each class' initializer, we can define a pre-ordering $A \triangleright B$ to mean "the initialization of class A forces the initialization of class B." If we use $initialized(A)$ to symbolize that class A is initialized, we have two additional rules we can use to infer whether a class is initialized:

- If $initialized(A) \wedge A \triangleright B$, then $initialized(B)$
- If $\neg initialized(B) \wedge A \triangleright B$, then $\neg initialized(A)$

To take advantage of these rules, the holographic VM architecture performs a conservative analysis of the effects of each class initializer method encountered while translating bytecode. We use an abstract interpretation [21] similar to the type inferencing algorithm used by JVMs to verify bytecode, where the abstract values are three-valued booleans indicating whether a value is a default value, is not, or could be either. The output of this data-flow analysis is both a three-valued

boolean for each static field and the set of classes the method's execution is guaranteed to force the initialization of. When it is necessary to check if a holographic class A is initialized, the class' static field values are compared with the static analysis results as described above. All classes B for which $A \triangleright B$ are also checked recursively, and if any are definitely uninitialized A is determined to be uninitialized as well.

This analysis is sound but not complete: classes may still be encountered for which the rules above are not enough to infer whether it is initialized. We further observe, however, that many class initializers are idempotent, in that they may be executed more than once without any additional side-effects. This means they can safely be run on classes that may already be initialized. The architecture thus includes another pluggable mechanism for users to mark specific classes as having idempotent class initialization. In Section 5.3.3 we provide evidence that this necessity should be relatively rare. It is also possible for a class to have non-inferable initialization status and a non-idempotent initializer, but we have yet to encounter such a case.

5.2.4 Concurrency

Our holographic VM implementation is currently limited to single-threaded execution, but there are no assumptions in the architecture that would prevent concurrent holographic execution. Like the JDI model, executing code in a holographic VM happens in the context of a specific thread mirror from the heap dump, and uses a dedicated native thread in the host VM to execute the translated bytecode. The semantics are identical to invoking a method on a paused thread while debugging a live process.

In order to support multiple native threads simulating multiple holographic thread executions, the data structures used in the mutable object graph layer described in Section 5.1.2 simply need to be replaced with their appropriately synchronized equivalents: replacing `HashMap` instances with `ConcurrentHashMap` instances, for example. The synchronization overhead will have a negative impact on performance, which should be the subject of future evaluations, but this will enable more complex post-hoc application simulation.

5.3 Evaluation

This section evaluates two primary research questions:

1. To what extent does the holographic VM architecture improve on the reflection-based approach to heap dump analysis?

2. Is holographic execution responsive enough for a typical heap dump analysis scenario?

5.3.1 Case Study: Diagnosing a Memory Leak

To evaluate the feasibility and utility of object holographs, we augmented the Eclipse MAT to leverage them as much as possible and then used the modified tool to diagnose a real world memory leak contributed by an end user.

Extending the Eclipse MAT

A large portion of the Eclipse MAT user interface centres around navigating and summarizing the object graph through predefined parameterized queries, some of which are directly analogous to source-level operations; "Extract List Values," for example, iterates through a list's entries in the same way as list iterator objects do. Our primary augmentation of the tool was to define two additional generic queries whose implementation used holographic execution.

The first is "Evaluate Expression," which parses and evaluates a given code snippet in the context of the objects selected in the tool. This is accomplished by adapting a holographic VM to the JDI and reusing the implementation of the Eclipse debugging UI. It supports either evaluating the expression once for each selected object or collecting all selected objects into a single `Collection` through a boolean-valued "aggregate" parameter.

The second is "Load and Run Code," which evaluates the contents of a specified method from a given class file on disk. This is accomplished by using holographic execution to create a new class loader instance, pass the class bytecode into the appropriate method to make the class loader define the new class, and then actually invoke the target method. This allows users to define more complicated

```
1  public static Map<String, Integer>
2      findDuplicates(Collection<CPPASTName> names) {
3
4      SortedMap<String, Integer> counts =
5              new TreeMap<String, Integer>();
6
7      for (CPPASTName n : names) {
8          String nKey = n + " - "
9              + Arrays.toString(n.getNodeLocations());
10
11         Integer count = counts.get(nKey);
12         if (count == null) {
13             count = 0;
14         }
15         counts.put(nKey, count + 1);
16     }
17
18     return counts;
19 }
```

Figure 5.2: Analysis code used to diagnose the Eclipse CDT memory leak bug

queries via additional code compiled against the original application binaries. This query also supports the same "aggregate" parameter.

We also replaced several existing queries with equivalent versions that used holographic execution. The "Extract List Values" query, for example, was reimplemented to invoke the `iterator()` method on any collection and use the result to iterate over the collection's elements. This not only increased the generality of the resulting queries, in this case allowing it to work on any `Collection` rather than only specific `List` subtypes, but also enabled them to accept newly created holographic objects as well as existing heap dump objects. Replacing these reflection-based query implementations also eliminated thousands of lines of code, showing that holographic execution also simplifies tool development.

Debugging Experience

The Juno release of the Eclipse C and C++ Development Tools (CDT) contained a memory leak[2]: indexing a large project caused the Eclipse runtime to exhaust all available memory, where the same project was successfully indexed in previous versions of the CDT. The user reporting the bug was able to upload a 1 gigabyte heap dump from the time of failure, but because the project that caused the error contained proprietary code they were not allowed to provide the actual project source. This hindered attempts by the CDT contributors in the following months to reproduce the problem, despite multiple other users reporting the same bug.

The CDT contributors were able to determine that approximately 80% of the heap was retained by over 1.8 million instances of the class `CPPASTName` and their related child objects. This class is used to represent unique occurrences of symbols in C++ source code after preprocessing, and the bug reporter's estimate of the actual number of such symbols in their code was smaller by a factor of six. Our initial theory for the memory leak was that the indexing process was creating multiple duplicate name objects representing the same locations in the source code. A straightforward way to investigate this theory is to iterate over all of the name objects and group them by their locations, in order to detect multiple names from the same location.

Obtaining the necessary bytecode for the relevant classes in the uploaded heap dump was not difficult in this case: we only required the appropriate versions of the Java 6 JRE and the Juno Eclipse distribution. We then authored a small helper method, built against the matching version of the CDT source code, which iterates over a sequence of `CPPASTName` objects and populates a map keyed by a string representation of their locations. See Figure 5.2 for the relevant source code. This analysis would be very time-consuming to implement using reflection: although the `CPPASTName` class has a field for storing its location, it is lazily calculated on request using several related datatypes, and so for the majority of the objects in the heap dump this field contains `null`.

We executed this code using the "Load and Run Code" query described above on the first 100,000 `CPPASTName` objects in the heap dump, resulting in a new

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=400073

holographic `HashMap` object. To examine its contents in the Eclipse MAT UI, we executed a holographic query to extract the key and value pairs from any `Map` instance. The results confirmed our theory that there were many sets of duplicates, in many cases over a dozen symbols with the same name and location.

Given that many of the most duplicated symbols were from a common library, our next theory was that the indexer was creating a separate symbol instance every time a header file was included. We selected one of the most duplicated symbols and began to test this new theory by writing code to print out the path of include declarations for each. The first step was traversing the parse tree to find the compilation unit containing each name, which we achieved using the "Evaluate Expression" query on the string `"getTranslationUnit()"`.

We were surprised to find that each symbol came from separate compilation units. Executing `"getFilePath()"` on each revealed that they were all for the same source file. From this point it was relatively simple to use existing MAT queries to find the references keeping the extra parse trees from being collected by the garbage collector, in particular a thread local that was not cleared after use. This analysis was presented on the online bug report, and a fix was submitted shortly after by one of the project contributors.

5.3.2 Performance

To determine whether holographic execution is performant enough for its intended use, we created a test harness that executes the `toString` method on every object in a `VirtualMachineMirror`, measuring the time taken to return from the invocation. This benchmark was chosen because it is easy to implement and applicable to any Java codebase, and yet exercises a surprising amount of code; even very simple implementations of `toString` are often only the tip of the iceberg when all of the methods that are ultimately used in their control flow are included.

We ran our benchmark against three sample applications. `jre_only` is a stub application including only an empty main class, for the purpose of benchmarking only the contents of the JRE. `tomcat` is the Apache Tomcat web server, version 7.0.37, after serving the initial welcome page. `eclipse` is the Eclipse IDE, build 20130614–0229, with a minimum of plugins installed in order to keep the total

	Application	jre_only	tomcat	eclipse
	Classes	456	2657	7610
	Objects	2249	46387	99452
Live VM	Avg. toString time (ms)	15.9	25.9	33.2
	Max. toString time (ms)	1748	22041	80234
	Std. Dev. toString time (ms)	74.4	279.3	512.6
Holographic VM	Prepare time (s)	44	171	340
	Avg. toString time (ms)	5.4	2.5	7.4
	Max. toString time (ms)	1279	8804	55867
	Std. Dev. toString time (ms)	38.7	78.7	325.6

Table 5.1: Results of executing Object.toString on every object in a VM, comparing performance on a holographic VM versus a live VM via the JDI

class and object count manageable.

For each sample application, we used the JDI to connect to and pause the live process, captured a single heap dump, and then ran the benchmark against both the live process and the snapshot. We used the performance of remote execution on a live process as the baseline, measuring the performance of holographic execution as a kind of overhead compared to this baseline. These experiments were performed on a MacBook Pro laptop with a 2.4 GHz Intel Core 2 Duo CPU and 8 GB of RAM, running Mac OS X version 10.7.4. Table 5.1 presents our results.

Although the time to translate the bytecode for all classes in the VMs is significant, once loaded the local holographic VM actually executes these methods faster than the remote process. In all cases the minimum `toString()` time was 0ms, as several classes define the method to simply return a constant or recalculated value, and hence return essentially instantaneously.

Since there is no convenient method for uniquely identifying objects in Java, and hence no convenient way to correlate the object mirrors in the two different VM mirrors, the data are analyzed as two independent sets. It would be instructive in future work to develop an algorithm for matching objects between VMs, possibly using structural comparison or raw memory addresses, in order to match times and analyze the average overhead.

We were surprised to discover that holographic execution is actually faster in all cases than remote execution using the JDI. We suspect this is due to the fact that the JDI relies on inter-process communication to pass values between the target and source VMs, whereas a holographic VM's state resides in memory with the caller, and for relatively simple objects this invocation overhead is greater than the time needed for the execution itself. A future evaluation could connect the JDI to a remote holographic VM instead of a local one to normalize this difference, but this would require additional engineering to accomplish. In addition, since a holographic VM does not have to reside in a separate process, the lower latency of reflective calls is in fact observed in tooling, although traded off by the overhead of keeping the object graph in memory locally.

The most expensive aspects of holograph execution are fetching the bytecode for the original classes as described in Section 5.2.1 and translating that bytecode to produce hologram classes as described in Section 5.1.3. When the extra step of loading a heap dump and preparing the holographic VM are included, the total times to run the benchmark on either a live or dead process are similar. Since the results of this process for each class in the heap dump are cached on disk, however, successive analysis runs can avoid this processing time. Our experience shows that the translation time is consistently about 5 seconds per megabyte of class file content. Preprocessing the entire JRE, which consists of over 20,000 classes and over 60 MB of bytecode, can be done in just under five minutes.

5.3.3 Completeness

The major limitation on completeness in this system is the possibility that the execution of useful code could encounter unsupported or illegal native methods. Our experimentation has initially targeted the Mac OS X distribution of Java 7 release 5, and for every native method in that JVM's runtime we have either provided a mirror-based alternate implementation or explicitly determined that it requires illegal access to the external environment and marked it as forbidden. Section A.1 lists our categorization of these native methods.

Our implementation currently includes 99 alternative implementations of native methods, with a total of 1443 source lines of code. This serves as a rough eval-

uation of the effort involved in supporting a particular VM implementation. Only those methods in the `sun.misc.Unsafe` class are specific to the exact JVM implementation we used, as they involve raw memory addressing that depends on the exact memory layout of its objects. There are also a handful of platform-specific classes such as `UNIXFileSystem` that contain native methods, but this is only a superficial platform dependency since the actual MNMs for such methods are only trivially different. 187 legal native methods are not yet implemented in our prototype, as they were not encountered in our experiments, but these are all either trivial variations on other implemented methods, such as alternatives for different primitive types, or alternative ways of accessing the reflective properties already supported by the mirrors API.

The other limitation on completeness is the possibility of encountering classes in the heap dump with indeterminate initialization status. In the process of supporting our experiments we encountered only two[3] such classes. We explicitly marked these as having class initializers that were safe to re-run, implying they could safely be executed even if they may have already run.

5.4 Related Work

5.4.1 Mirror-based Behavioural Intercession

Prior work has examined the idea of customizing the behaviour of a language's objects via implicit mirror implementations. Such objects have been called *mirages* [46] or *virtual values* [8], and have largely been studied in the context of behavioural intercession, or augmenting or replacing behaviours on existing objects. Holographic objects are similar in implementation, but focus instead on reproducing the behaviour of base objects with no actual base object available in the runtime to provide the base behaviour. In our case, behavioural intercession is limited to replacing native methods, with the specific requirement of *not* deviating from base behaviour, and is not exposed to developers that use the library.

In addition, prior work has presented implementations of such objects in dynamic languages, and doing the same in a statically-typed language such as JVM

[3] `java.lang.reflect.Modifier` and `java.security.KeyFactory`

bytecode without requiring a custom language runtime presents fundamental challenges that affect the design of the interfaces to those objects. The Jikes Research Virtual Machine [5] also implements the behaviour of a JVM on another JVM, and the majority of its code is ordinary Java. Its object model is not intended to be pluggable, however, and although it could be made so this would not necessarily be any easier than customizing any other JVM implementation.

Lorenz and Vlissides [40] describe how pluggable reflection enables more flexible language tools, using a documentation generator and an object-to-component generator as examples. A holographic language runtime represents another client of a language's pluggable reflective API, and allows the language's implementation itself to be decoupled from the representation of its runtime state. Unlike Lorenz and Vlissides' examples, holographic execution requires a reflective API that represent computation and not just code, a distinction clarified by Bracha and Ungar [13].

5.4.2 Reproducing Past State and Behaviour

The idea of checkpointing and resuming execution recurs in several contexts, notably in operating system or hardware virtualization. Some language runtimes also support resuming from a snapshot [1], and Java itself includes a small amount of this behaviour in its shared memory class file cache feature. In all of these cases, the restored process is a normal, unrestricted instance, and care must be taken to ensure that invalid references to the outside world are not created. By contrast holographic execution as described in this work is intended to support diagnosis and analysis of the state of a system in the past. It requires no foresight, but at the cost of restricting additional execution and hence not truly restoring a dead process. It also does not currently support resuming execution from the exact time of the snapshot, although this is conceivably possible if a more precise snapshot such as a core dump were used, along with techniques for recreating the captured call stack via program slicing [67].

5.4.3 Heap Dump Analysis

Several other approaches have been used to analyse heap dumps, usually in the context of identifying the source of memory leaks. Maxwell et. al. [44] use graph mining to locate potential leak candidates. As we have illustrated in Section 5.3.1, holographic execution is a complementary technique which does not preclude the use of the reflective API on a heap dump. For example, the aforementioned work includes a case study of a memory leak in a scripting language parser. The graph mining technique identifies a non-standard linked-list implementation containing a long series of regular expression matches, which is helpful but does not fully diagnose the root cause. We postulate that applying holographic execution to print out the semantic contents of this list could be extremely useful in diagnosing the actual source of the leak.

5.4.4 Static Code Analysis

Kozen and Stillerman [35] use a static analysis of class initializers similar to ours to initialize classes eagerly, in order to improve startup performance and catch errors earlier. Their algorithm ignores initializer effects with respect to static fields, but is flow-sensitive and hence calculates a more precise definition of initialization dependencies than our current implementation. Integrating their approach in the future may improve the success rate of our algorithm and hence reduce the number of initializers that must be marked safely repeatable.

5.5 Summary

We have presented an architecture for holographic objects, which enables restricted execution starting from the state captured in a heap dump. We have shown that this architecture supports analysis that is simple, type-safe, robust, secure, and familiar. It provides a good fit for analysis that depends on application-specific semantics, but also complements meta-level analysis by supporting a hybrid approach. Our experience with our implementation for Java bytecode suggests that this architecture can be effectively realized without having to customize a JVM. We have presented evidence that holographic execution is competitive with live execution and hence is feasible for heap dump debugging and analysis. We have also applied this proto-

type implementation to diagnose a real-world memory leak bug that went unsolved for several months.

Chapter 6

Retroactive Weaving for AspectJ

This chapter describes the architecture for our retroactive weaver for the AspectJ AOP language, which builds on the architecture for holographic virtual machines presented in the previous chapter, and provides evidence for the effectiveness of this architecture through case studies and an evaluation of performance. This in turn provides evidence for the feasibility and effectiveness of the time-travel programming paradigm for mature, full-featured programming languages.

6.1 Architecture

At a high-level, we implement the process of retroactive weaving for AspectJ by combining three large and orthogonal components. The first component is the interface to the underlying execution recording, adapted to implement the generic VirtualMachineMirror (VMM) interface described in Section 5.1.1. This hides the details of whichever recording system is used, and indeed our evaluation applies this architecture to two different systems. The second component is a holographic virtual machine that wraps the first component and implements the same interface, providing the ability to perform additional execution on top of the recording as described throughout Section 5.1. The last component, the primary focus of this chapter, is a *reflective aspect weaver*, which uses the metaprogramming facilities of the `VMM` interface to trap dynamic join points as they occur in the recording and execute matching aspects accordingly.

This architecture allows the weaver to focus only on the concern of dynamically weaving aspects against a VMM, whether that program represents live execution or a recording, and only the holographic VM is concerned with emulating the retroactive execution specified by aspects. The modular approach helps to reduce the complexity of this system, by separating the execution recording technology and the AOP language semantics as independent concerns. The key to achieving this modularity is the VMM interface, which is a reflective definition of the target programming language.

6.2 Events and Intercession

In Chapter 5, a VMM was only used to represent a snapshot of a running program at one instance in time. To implement retroactive weaving for an execution recording, this model must be extended to represent the behaviour of a program as it executes over time. The necessary interface is somewhat similar to the reflective interface a JVM provides for implementing debuggers, and indeed one of the scenarios we use to evaluate retroactive weaving in Section 6.7 below involves adapting the Java Debugging Interface described in Section 2.3 to the VMM interface.

The key new operation is the ability to register callbacks for runtime events, such as method calls and field accesses, so that when events occur they are first passed to those callbacks before they actually occur in the runtime. While the VMM invokes such callbacks, the state of the runtime is frozen so that callback bodies may read any necessary contextual state. Moreover, the callbacks are allowed to modify the events, effectively changing runtime behaviour. This augments the runtime with powerful metaprogramming facilities.

VMM instances are initially created in a paused state. Clients add initial callbacks as above, and then invoke a `resume` method that conceptually continues execution. For an execution recording, there may be no actual resumed execution. The interface only requires that each requested event is delivered to the registered callbacks in the correct order, with the VMM state emulating the context in which each event occurred.

The VMM interface also includes a handful of operations for modifying the structure of classes, such as adding new fields to an existing class, or declaring that

a class implements additional interfaces. These are used by the reflective weaver to implement intertype declarations, as well as to attach extra state for the internal representation of features such as `cflow` pointcuts.

Note that this interface is strictly more powerful that the reflective capabilities of the Java or AspectJ programming languages themselves. As observed in Section 2.2, it is not possible for base-level Java code to intercept and modify method invocations using the reflective classes included in the Java standard runtime libraries, such as `java.lang.Class` and `java.lang.Method`. It is possible, however, to provide at least some of these capabilities when adapting various execution recordings to the `VMM` interface. The wrapping holographic VM then provides some of the missing metaprogramming functionality as well, just as it already fills in missing state as described in Section 5.2. The holographic VM implementation also raises events for holographic execution itself via hooks in its generated hologram class bytecode, which ensures consistent semantics in line with the definition of retroactive execution.

6.3 Reflective AspectJ Weaver

Our reflective weaver uses the metaprogramming facilities of the `VMM` interface described above to achieve the semantics of AspectJ aspects. The weaving process registers callbacks that perform additional execution or modify the events themselves, such that the observed behaviour is equivalent to static code modification.

Note that the reflective weaver is not a runtime weaver for AspectJ. Although the flexibility and power of metaprogramming potentially supports operations such as dynamically enabling and disabling aspects, those semantics are not included in the definition of AspectJ, as noted in Section 2.5. Since the reflective weaver is an alternative implementation of the same semantics of AspectJ, the weaver's use of metaprogramming is carefully controlled to avoid exposing any additional behaviour.

The event requests which trigger the execution of aspect code are all installed on the recorded `VMM` before actually resuming the recorded execution. Other requests may be dynamically enabled or disabled as the recorded execution proceeds, but only as needed to optimize the weaving process or implement late binding. For

example, when a new class is defined in the recording it may be necessary to create new event requests to trap events in that class' code. The holographic VM also registers its own internal callbacks for specific events to collect data necessary for correct holographic execution.

Reflective weaving borrows much of the load-time configuration implementation described in Section 2.5 as well, in particular support for Extended Markup Language (XML) files that explicitly name aspects that should be woven. These may be hand-written by developers, but are also automatically generated by the AspectJ compiler when compiling aspects to their binary form for load-time weaving.

The high-level process for retroactive weaving is:

1. Load an execution recording, adapted to implement a `VMM`;
2. Instantiate a holographic VM around it;
3. Read the list of names of aspects to weave from the relevant XML files;
4. Load the requested aspects by name in the holographic VM;
5. For each aspect:
   (a) modify classes in the VM according to any intertype definitions, and
   (b) register callbacks to execute advice at matching join points;
6. Resume the holographic VM

Note that the reflective weaver supports a `-XweaveCoreClasses` option in the XML configuration which `ajc` does not. This is necessary because the reflective weaver, unlike `ajc`, is able to implement the semantics of weaving aspects against classes in standard Java class libraries. Many existing AspectJ aspects assume the weaver only affects user-provided classes, or those loaded by a specific class loader. If these are woven against core classes using the reflective weaver, they often become extremely slow or even incorrect. However, some of the aspects in our evaluation case studies (see Section 6.6) do require access to core classes.

In addition, avoiding retroactive side-effects in the core libraries using the technique described in Section 3.4 requires weaving against core classes. Providing this option allows clients to scope their aspects on a per-library basis.

6.3.1 Events as Join Point Shadows

The core of the `ajc` weaver already operates on an abstracted representation of the AspectJ language in order to support both compile-time or binary weaving. Creating support for reflective weaving largely reduces to creating another implementation of this abstraction backed by the mirrors API. Recall from Section 2.5 that the `ajc` weaver is architected around the idea of join point shadows, which are locations in AspectJ code (source or binary) that give rise to dynamic join points. Because the `ajc` weaver materializes the semantics of AspectJ by transforming code into Java byte code with equivalent behaviour, the primary function of the weaver is to iterate through all join point shadows in the code and test for advice whose pointcuts match those shadows. When a match is found, shadow mungers are applied that transform the bytecode as described above.

The reflective weaver repurposes this mechanism and reuses much of the `ajc` implementation by adapting dynamic runtime events as join point shadows. It is straightforward for a runtime event from a `VMM` instance to expose at least as much metadata as the static area of code that gave rise to the event. For a method call, for example, this includes the declaring class, method name and argument types, as well as less fundamental properties such as the source file name and line number of the method call.

Shadows backed by a runtime event also support actually evaluating the shadow. This is exposed as a closure which can be replaced on the underlying event. For a runtime event, a shadow munger will modify the event by wrapping the exposed closure to add additional computation before, after, or around the shadow.

6.3.2 Efficient Event Requests

Given a pointcut, the weaver must create a minimal but sufficient set of event requests to cover all matching dynamic join points. A valid but extremely inefficient implementation of reflective weaving would be to register callbacks for all possible

events, since the generic pointcut matching logic will filter out irrelevant events. It is possible to optimize the weaving process considerably, however, if the pointcuts are partially and conservatively translated into event filters. The underlying execution recording mechanism will frequently support some of these filters natively and hence avoid some of the substantial overhead of detecting, instantiating and handling unnecessary events. The details of implementing these filters in the underlying `VMM` are examined in Section 6.4.

All relevant shadow evaluations are therefore handled in a two-phase approach: filters are used to trap an over-approximation of all relevant events in the recording, and all resulting events are checked against the relevant pointcuts before actually applying their shadow mungers. This two-phase approach mirrors the implementation of compile-time or load-time weaving, where shadows are matched statically to determine locations in code that may produce matching join points, but dynamic testing logic often has to be inserted to match a portion of the pointcut at runtime.

AspectJ pointcuts are formed from a combination of several possible patterns. They can be divided into four categories:

1. Kinded, such as `call` and `set`;
2. Scoping, such as `within`;
3. Contextual, such as `this`; and
4. The pointcut combinators `&&`, ||, `!` and `cflow`

An early stage of pointcut concretization replaces `cflow` pointcuts with a combination of explicit control flow tracking and pointcuts that refer to that state. The remaining pointcut combinators resemble logical operators: `&&`, || and `!` for "and", "or" and "not" respectively. Pointcuts thus have an equivalent concept of Disjunctive Normal Form (DNF). Rewriting a pointcut in DNF is already implemented and used by the `ajc` weaver to optimize the shadow matching process. In our implementation it is helpful to lift all uses of || to the top-level, since event requests for each conjunctive clause can be extracted independently.

The pseudocode for extracting a sufficient set of event requests for an advice declaration is as follows:

```
Reduce the pointcut to disjunctive normal form
result ← ∅
For each conjunctive clause:
    Separate the kinded designators from the others
    Calculate the set of kinds that will match those designators
        If there are no kinded designators, match all join point kinds
    For each matching kind:
        request ← a new mirror event request of that kind
            For example, a `set` corresponds to a `FieldMirrorSetRequest`
        For each other designator in the clause:
                and signature pattern in each kinded designator:
            If the kind does not support this pattern
                    (e.g. `* MyClass.foo(int)` for a `set`):
                Continue with the next kind
            Otherwise:
                If the mirror event request supports this pattern
                    Add a corresponding filter to the `request`
        Add request to result
return result
```

Note that if a conjunctive clause contains two kinded designators of different kinds (e.g. `set` and `call`), the pointcut will have no matches, since kinds define a partition on join points and hence no join point will ever match more than one kind simultaneously.

It is important to optimize the implementation of `cflow` pointcuts to drastically reduce the number of events raised by the underlying execution recording. For all pointcuts that contain `cflow` expressions, the event requests created to match their join points are initially disabled. They are only enabled when a join point that matches the argument to the `cflow` expression is entered, and then disabled when the join point exits. This is especially critical given the idiom of using `cflow(adviceexecution())` to avoid retroactive side-effects, since many of the relevant join points will occur very frequently in the base program. See Section 6.7.1 for examples of such aspects.

6.4 Execution Recordings

Execution recording technologies vary in terms of how much information they explicitly record and how much is implicit and inferred or reconstructed at consumption time. To evaluate the effectiveness of our implementation of retroactive weaving, we also adapted two implementations of execution recording to the `VMM` interface: one based on a database of events and snapshots, and one based on deterministic replay. The implementations we chose represent two contrasting approaches to execution recording, and hence are intended to help measure the behaviour of retroactive weaving over the range of all possible approaches to execution recording.

Many recent approaches to execution recording use a combination of these two techniques, including Pothier's more recent work on omniscient debugging [55]. Such systems will often record multiple snapshots of system state, for example, and then use partial replay to determine state in between them. By choosing highly contrasting backends, our evaluation is intended to provide the maximum evidence for how the effectiveness of retroactive weaving varies with the underlying approach. Implementing two very different backends also provides evidence that the `VMM` interface is sufficiently generalized, and hence that reflective weaving will be applicable to other possible execution recording technologies.

6.4.1 Events Database

Our first execution recording adapter is based on the tracing and database backend of the Trace-oriented Debugger (TOD) presented by Pothier et al. [56]. At its heart, this approach closely resembles the tracing implementation shown in Section 4.6.1, since at a high level the events database is simply a sequence of instantaneous events. Each event contains a synthetic timestamp, where events with the same timestamp semantically occurred simultaneously in the original execution.

The TOD backend exposes a querying interface similar to that of a relational database. The trace database provides an interface for creating a cursor over a subset of events from the trace sequence. Filters can be applied to pick out only those events that occurred within the source of a particular class, on a specific object, and so on, and filters can be combined with conjunction and disjunction operators

to create compound filters. Events are aggressively indexed as they are recorded to make these filters efficient to implement. The trace database also provides various interfaces for querying the state of objects and arrays at any given timestamp. These operations are used by the omniscient debugger UI to step forwards and backwards in time, and to present a debugger-like view on the state of the recorded program at any given point in time.

In the TOD `VMM` adaptor, each callback request is translated to an event cursor, and each supported filtering operation translates almost directly to cursor filters. A single stream of matching events is easy to produce using a merge sort process similar to that described in [56]. The state of the adaptor includes a current timestamp, effectively slicing the data to the state at this timestamp, and the `VMM` interfaces for querying state are implemented by passing this fixed timestamp to the database query methods. The `resume` operation semantically replays the recorded program by iterating through the result of merging the cursors for all requests. Each time the cursor moves to the next event, the `VMM` timestamp is updated to that of the event. Then each applicable callback is invoked on the event. The adaptor only reads state in a forwards direction, and hence does not take advantage of TOD's ability to debug backwards in time, but it may skip over large periods of time as it travels forwards and queries only relevant events.

6.4.2 Deterministic Replay

Our approach to retroactive weaving based on deterministic replay is to enable debugging on the DR process, attach a debugging connection to that process within a second JVM using the Java Debugging Interface (JDI), and adapt that JDI connection as a `VMM`. This ensures isolation from whatever DR mechanism is used, at the cost of decreased performance due to the inter-process communication. This cost is mitigated somewhat by caching data within the holographic VM that wraps the JDI adaptor.

The callback API in the `VMM` model is straightforward to implement using the JDI's similar event request mechanism. Although the `VMM` model supports concurrency via threads, the JDI adaptor pauses all threads within the debugged VM whenever an event is raised. Assuming asynchronous threads rather than concur-

rent threads in the VMM model greatly simplifies the holographic VM implementation, especially with respect to ensuring consistency in the aforementioned caching.

We assume that implementing the API for event callbacks does not cause perturbation of the underlying process. This is true as long as the underlying replay mechanism is robust enough to be unaffected by the pauses in execution and additional reads of program state caused by event callbacks.

For the purposes of our evaluation we use the LEAP system for deterministic replay of multi-processor systems [31]. Like many software-based approaches to DR, LEAP works via static analysis and bytecode instrumentation of the source program. It produces both a record and replay version of the source program; the former records thread ordering data which the latter takes as input in order to reproduce the recorded execution. Our experimental setup is therefore to connect to the running replay process as the remote source of program state as above.

6.5 Soundness

Performing some operations may interfere with execution recording replay, because they would perturb the underlying VM such that there is not enough information to continue the simulation. It is the responsibility of the execution recording adaptor to guard against operations that would invalidate the recording, since the precise definition of which operations are illegal will vary depending on the exact recording mechanism.

The execution recoding adaptors are configured to disallow operations that modify the VM's state, such as setting fields and invoking methods. If such operations occur in holographic execution, the adaptors will throw an instance of an exception class named `IllegalSideEffectError`. This class is declared as a subclass of `VirtualMachineError`, a base class used internally by JVM implementations to indicate low-level failure conditions client code should not be expected to catch and reasonably handle, such as exhausting memory or encountering internal VM bugs. We intend clients of the VMM interface to catch `IllegalSideEffectErrors` instead, at the meta-level.

6.6 Case Studies

This section outlines the background and motivation behind the examples of AspectJ aspects we used to evaluate retroactive weaving for AspectJ. Our evaluation targets both the effectiveness of the technique for the AspectJ language in particular and our weaver implementation. The former is demonstrated by reusing existing aspects, or by presenting alternate versions of dynamic analysis expressed as clear and concise aspects. The latter is evaluated by measuring the performance penalty of evaluating aspects retroactively, and by approximating the additional development effort necessary to avoid illegal retroactive side-effects.

We have endeavoured to identify several classes of aspects that are well-suited to retroactive weaving, and chosen one concrete aspect to represent each class. We have in general preferred generic aspects, meaning those that are applicable to any arbitrary Java or AspectJ program, so that we can use the same set of base programs to record and retroactively weave. This helps to keep a consistent baseline for comparing these different classes of aspects.

We present these case studies in roughly increasing order of complexity. The following table summarizes which AspectJ and holographic execution features each case study makes use of.

Table 6.1: Summary of case studies and the AspectJ features they use

	contract	tracing	racerj	leaks	heap
stateful	y	y	y	y	y
cflow	y	y	y	y	y
around					
toString	y	y			
recorded threads			y	y	y
retroactive threads				y	y
type reflection					y
control reflection				y	

The features referenced by the table are as follows:

stateful: Whether the analysis code maintains its own state between join points.

cflow: Use of the `cflow` pointcut combinator.

around: The inclusion of around advice. Note that while around advice does

not appear in the source of any of the case studies, it is used heavily in the aspects used to avoid illegal retroactive side-effects as illustrated in Section 6.7.1.

toString: Use of the `Object.toString` method to output objects in a human-readable form.

recorded threads: Supporting the analysis of multiple threads in the original execution.

retroactive threads: Analysis that spawns its own additional threads.

type reflection: Analysis that uses reflective methods to analyze the structure of types that appear in the recording (e.g. `Class.getDeclaredFields`).

control reflection: Analysis that uses reflective methods to analyze the control flow of the recorded program (e.g. `Thread.getStackTrace`).

6.6.1 Contract Verification

A commonly cited application for aspects is modularizing contracts when using the Design by Contract (DbC) paradigm [45]. Some argue that using aspects in this way fails to maintain the advantages of DbC [10], although there is recent work on AOP languages intended to mitigate this [57]. However, contracts expressed as aspects are also good candidates for retroactive weaving because of the pluggability of aspects. They are often too expensive to leave enabled in production code, but may offer valuable insight into the root cause of unexpected behaviour when evaluated against an execution recording, where raw performance is less important.

The example we chose is a simple, lightweight aspect that detects resource leaks. It leverages the semantics of the built-in `java.io.Closeable` interface: any class which implements this interface represents a resource that should be released when the object is no longer needed. Newer versions of the Eclipse IDE will provide compiler warnings for objects that are never closed, but the static analysis the warning is based on cannot detect resource leaks for objects that escape their local scope. The aspect in Figure 6.1, however, will detect such leaks dynamically by capturing all instances of classes that implement `java.io.Closeable`, tracking when they are closed, and reporting on those that were never closed when the program terminates.

```
1  public aspect UnclosedCloseables {
2
3      private final Set<Closeable> unclosed = new HashSet<Closeable>();
4
5      after() returning(Closeable closeable): call(Closeable+.new(..)) {
6          unclosed.add(closeable);
7      }
8
9      after(Closeable closeable): call(* Closeable.close())
10                                 && this(closeable) {
11         unclosed.remove(closeable);
12     }
13
14     public UnclosedCloseables() {
15         Runtime.getRuntime().addShutdownHook(new Thread() {
16             public void run() {
17                 System.out.print("Unclosed closables: " + unclosed);
18             }
19         });
20     }
21 }
```

Figure 6.1: A contract verification aspect

6.6.2 Tracing

The next case study is the quintessential tracing example described in Section 2.5. While relatively simple, the most sophisticated version of this aspect makes use of several AspectJ features that exercise the overall implementation well:

- Invoking the `toString` method within advice, implying a non-trivial amount of retroactive execution;
- A `cflow` pointcut to avoid infinite recursion when calling `toString`;
- References to the special form `thisJoinPointStaticPart`, which exposes metadata about the join point; and
- The use of persistent state between advice invocations to implement tracing indentation

See Listing 6.1 for an excerpt of the output of this aspect.

Listing 6.1: An excerpt of output from the AspectJ tracing aspect

```
1   --> double tracing.Circle.perimeter(): Circle radius = 2.0 @ (3.0, 3.0)
2   <-- double tracing.Circle.perimeter(): Circle radius = 2.0 @ (3.0, 3.0)
3  c1.perimeter() = 12.566370614359172
4   --> double tracing.Circle.area(): Circle radius = 2.0 @ (3.0, 3.0)
5   <-- double tracing.Circle.area(): Circle radius = 2.0 @ (3.0, 3.0)
6  c1.area() = 12.566370614359172
7   --> double tracing.Square.perimeter(): Square side = 1.0 @ (1.0, 2.0)
8   <-- double tracing.Square.perimeter(): Square side = 1.0 @ (1.0, 2.0)
9  s1.perimeter() = 4.0
10  --> double tracing.Square.area(): Square side = 1.0 @ (1.0, 2.0)
11  <-- double tracing.Square.area(): Square side = 1.0 @ (1.0, 2.0)
12 s1.area() = 1.0
13  --> double tracing.TwoDShape.distance(TwoDShape): Circle radius = 4.0 @ (0.0, 0.0)
14    --> double tracing.TwoDShape.getX(): Circle radius = 2.0 @ (3.0, 3.0)
15    <-- double tracing.TwoDShape.getX(): Circle radius = 2.0 @ (3.0, 3.0)
16    --> double tracing.TwoDShape.getY(): Circle radius = 2.0 @ (3.0, 3.0)
17    <-- double tracing.TwoDShape.getY(): Circle radius = 2.0 @ (3.0, 3.0)
18  <-- double tracing.TwoDShape.distance(TwoDShape): Circle radius = 4.0 @ (0.0, 0.0)
19 c2.distance(c1) = 4.242640687119285
20  --> double tracing.TwoDShape.distance(TwoDShape): Square side = 1.0 @ (1.0, 2.0)
21    --> double tracing.TwoDShape.getX(): Circle radius = 2.0 @ (3.0, 3.0)
22    <-- double tracing.TwoDShape.getX(): Circle radius = 2.0 @ (3.0, 3.0)
23    --> double tracing.TwoDShape.getY(): Circle radius = 2.0 @ (3.0, 3.0)
24    <-- double tracing.TwoDShape.getY(): Circle radius = 2.0 @ (3.0, 3.0)
25  <-- double tracing.TwoDShape.distance(TwoDShape): Square side = 1.0 @ (1.0, 2.0)
26 s1.distance(c1) = 2.23606797749979
27 s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

6.6.3 Race Detection

Bodden and Havelund developed RACER [12] as an example implementation of an algorithm for detecting data races. They proposed three novel pointcuts to enable a whole class of aspects for analyzing concurrency bugs: `lock` and `unlock`, for picking out join points that acquire and release locks, and `maybeShared`, a scoping pointcut used to restrict other pointcuts to those involving data possibly accessed by multiple threads.

RACER consists of two aspects. `Locking` uses the `lock` and `unlock` pointcuts to track the locks held by a given thread and how many times each has been locked. `Racer` picks out possibly shared field accesses and drives the `RACER` algorithm's state machine. While the algorithm itself is relatively simple, its complete implementation in AspectJ is still over one thousand lines of code, and executing it retroactively is a reasonably heavy stress test of the architecture.

RACER is particularly compelling as a case study for retroactive weaving because its overhead is relatively high. The more that this high runtime cost can be pushed to the post-hoc processing environment, where raw performance is less of a concern, the more practical such analysis becomes. With similar goals in mind, Ansaloni et. al. [6] investigated the advantages of using buffered advice to offload some of this overhead to a separate thread. Unlike buffered advice, retroactive weaving does not require altering the specification and hence the semantics of the original advice, although it instead introduces the possibility of runtime errors due to illegal retroactive side effects as described in Section 6.5.

The `maybeShared` pointcut is used as an optimization to reduce the number of join points matched by the `Racer` aspect and hence the overall runtime of the algorithm. For the sake of simplicity, the retroactive weaver's implementation of this pointcut matches all join points; Bodden and Havelund note in [12] that this is a valid implementation, and moreover that the observed improvement in runtime speed due to their implementation based on a static thread-local objects analysis was negligible.

6.6.4 Memory Leak Detection

Section 5.3.1 provided evidence for the benefits of retroactive execution in diagnosing memory leaks based on snapshots. It is logical to investigate whether these benefits extend to execution recordings via retroactive weaving. Villazón et al. [66] and Chen and Chen [16] both experimented with using aspects to detect memory leaks, and illustrate that one of the advantages of this approach is the ability to track the dynamic location of each object construction, namely via stack traces.

We use Villazón's version for this evaluation since the full aspect's source is available for experimentation. This aspect maintains `PhantomReferences` to constructed objects to ensure they can still be collected. Before collecting such objects, the JVM adds the corresponding phantom references to a reference queue, which the aspect monitors. As references appear in the queue the aspect removes them from the set of potential leaked objects. This aspect therefore relies on the effectiveness of the JVM's garbage collection to eliminate false positives, and hence when woven retroactively stresses the fidelity of the holographic virtual machine.

To support this case study, we implemented a naive but semantically correct version of retroactive garbage collection. The key requirement is being able to determine when an object is not reachable either in the original execution or through additional retroactive references. Our approach is to rely on the host JVM's garbage collection to collect holographic wrapper objects when they become unreachable. We define `finalize` methods on a few key classes such that when a holographic object is collected the holographic VM can check to see if the corresponding object in the execution recording is reachable. The holographic implementation of the native `Runtime.gc()` method can then enqueue holographic references to any such objects.

By necessity, this split approach will not identify as many collectable objects as the native garbage collection. In the worst case, where an object is minimally reachable by a chain of n pairs of original and retroactive references, it will take n iterations to clear the chain of garbage. This is a valid limitation of a JVM garbage collector, however, as they are permitted to be non-deterministic and unpredictable. Villazón's aspect already accounts for the general imprecision of garbage collection by invoking `Runtime.gc()` and related methods multiple times until col-

lection reaches an approximate fixed point.

6.6.5 Profiling

Pearce et al. [52] investigated the effectiveness of implementing a general-purpose Java profiler using AspectJ. Their conclusions were that AspectJ was sufficiently expressive and efficient for reasonable implementations of several profiling use cases. They also named several limitations of the standard implementation of AspectJ at the time that impacted its suitability, primarily the lack of support for array construction and synchronization join points, and the lack of support for weaving classes in the standard libraries. To date, however, these limitations have mostly since been addressed: the requested join points have since been added, and several alternate weaving approaches have been proposed for weaving standard classes [66]. Reflective weaving is another such approach since it does not modify the original bytecode.

Their implementation `djprof` includes several aspects for gathering statistics on object lifetimes, heap usage, time spent, and time wasted. Profiling use cases that involve measuring actual wall time for execution are not ideal for retroactive weaving; modelling the current time during retroactive execution in a semantically consistent manner is challenging, although an interesting avenue for future work. However, analyzing execution in terms of operation frequencies or the size of data is just as effectively measured post-hoc. We chose to reuse the `HeapAspect` in particular from `djprof`, which estimates the total heap space allocated by each method. Our primary reason was to demonstrate the advantage of source compatibility in retroactive analysis: even though the aspect is generic and refers only to types within the standard libraries, it makes heavy use of the reflective methods in Java (and hence AspectJ) to estimate object sizes. These calculations could be implemented using other APIs for executions recordings, but using retroactive weaving means the same existing code can be used as-is.

6.7 Evaluation

This section evaluates two primary research questions:

1. What amount of effort, in terms of orders of magnitude, is necessary to adapt

an aspect to make it valid for retroactive weaving?

2. Is retroactive weaving responsive enough for a typical execution recording analysis scenario?

6.7.1 Adaptation Effort

This section quantifies the amount of additional programming necessary to adapt the selected aspects to retroactive weaving, using the idiom described in Section 3.4. One of our claimed benefits of retroactive weaving is the reuse of existing aspects as-is for post-hoc analysis. If substantial effort is necessary to avoid retroactive side-effects and hence ensure sound analysis results, it reduces the strength of this claim.

Many such effects occur within code that caches frequently requested data within the standard runtime libraries. Avoiding this class of effects is usually as simple as eliding the caching mechanism entirely, since retroactive weaving is not a performance-critical environment. Figure 6.2 contains an example of such code and advice for avoiding the retroactive side-effect. In this particular case, there is only a single, highly-localized operation to omit, since the hash field is private and only assigned to within this method, but in other cases it is necessary to use the `cflow` pointcut to apply a more precise scope.

Other illegal side-effects occur because aspects legitimately need to alter behaviour in ways that do not semantically interfere with the original execution. For example, three of the case study aspects we evaluated need to summarize and report analysis at the end of execution. They achieve this by registering threads to be run when the JVM is shutting down using the `Runtime.addShutdownHook` method, which adds threads to a static list. If called retroactively, adding to this list is an illegal side-effect.

The solution here is to declare an additional list for retroactive hooks on an aspect, and to define advice that augments the shutdown hook mechanism to also run hooks added to this second list. Figure 6.3 illustrates how this is done. This version advises the low-level implementation of the shutdown hook mechanism, at the cost of violating encapsulation. An alternative version could instead advise higher-level code, at the cost of some code duplication.

```
1  public class String {
2      ...
3      public int hashCode() {
4          int h = hash;
5          if (h == 0 && value.length > 0) {
6              char val[] = value;
7
8              for (int i = 0; i < value.length; i++) {
9                  h = 31 * h + val[i];
10             }
11             hash = h;
12         }
13         return h;
14
15     }
16     ...
17 }
18
19 public aspect JREAroundFieldSets {
20     ...
21     void around(): set(* String.hash) {
22         // Don't proceed(), just let it be recalculated every time
23     }
24     ...
25 }
```

Figure 6.2: Suppressing illegal side-effects with aspects

A key unanticipated technical challenge is that the process of loading the aspects themselves may also have illegal side-effects. These operations occur in the implementation of class loading within the standard libraries and not in the control flow of the base code it is loading. This creates a catch–22 situation where loading the aspects to avoid retroactive side-effects can cause those effects themselves. To avoid these side-effects, the holographic JVM implementation itself includes a small amount of low-level metaprogramming to install callbacks that relocate state and augment behaviour in much the same ways as described above. As discussed at length earlier, this code is more difficult to write and debug, but is limited to the bootstrapping phase of the holographic JVM.

In total, we encountered 21 instances of state the aspects attempted to retroactively modify in their control flow, 18 of which had to be elided through metaprogramming instead of advice. It is likely that this set of bootstrap workarounds

```
1  public aspect ShutdownHooks {
2
3      // Augments ApplicationShutdownHooks.hooks
4      private static IdentityHashMap<Thread, Thread> moreHooks
5              = new IdentityHashMap<Thread, Thread>();
6
7      // Add retroactive hooks to moreHooks instead
8      void around(Thread hook):
9              execution(void Runtime.addShutdownHook(Thread))
10             && args(hook) {
11         moreHooks.put(hook, hook);
12     }
13
14     after(): execution(void Shutdown.runHooks()) {
15         // Copy of ApplicationShutdownHooks.runHooks(),
16         // but referring to moreHooks instead.
17         Collection<Thread> threads;
18         synchronized(ShutdownHooks.class) {
19             threads = moreHooks.keySet();
20             moreHooks = null;
21         }
22
23         for (Thread hook : threads) {
24             hook.start();
25         }
26         for (Thread hook : threads) {
27             try {
28                 hook.join();
29             } catch (InterruptedException x) { }
30         }
31     }
32 }
```

Figure 6.3: Relocating illegal side-effects with aspects

are adequate for many other retroactive aspects as well, since they are enough to support safely loading any additional aspects necessary to avoid other side-effects. The complete list of workarounds can be found in Section A.2.

6.7.2 Results and Runtime

We now present a basic evaluation of the retroactive weaver as applied to each of the five case studies. We compared the output and runtime of the `ajc` load-time weaver against the retroactive weaver using either TOD or LEAP execution record-

ings. The results are presented in Table 6.2. All run times are reported in seconds, and are the average of five iterations with one warmup iteration (which has the side-effect of populating the holographic JVM bytecode caches as in Section 5.3.2).

Table 6.2: Execution time comparison of retroactive weaving with load-time weaving

	contract	tracing	racerj	leaks	heap
Base program	0.29	0.26	0.26	0.43	0.44
ajc load-time weave	3.68	3.60	8.80	3.95	3.37
TOD record	11.33	13.30	N/A	11.98	11.34
TOD retroactive weave	12.98	44.35	N/A	107.65	34.58
LEAP record	0.40	0.39	0.25	0.25	0.45
LEAP retroactive weave	16.52	28.10	151.73	104.30	26.01

From these results we can see that retroactive weaving typically performs between 3X to 20X slower than runtime weaving. The TOD-based retroactive weaver out-performed the LEAP-based weaver for the contract validation aspect because the aspect matched very few joinpoints in the base program. This is the situation where using an indexed execution trace offers an advantage, since the remaining execution events that do not match can be skipped over quickly.

The leak detector aspect caused the worst overhead for both execution recording adaptors at approximately 26X, but for very different reasons. As documented in Villazón et al. [66], the aspect is often more useful if woven with a weaver that supports core Java classes, and its corresponding base program contains a leak that is only detected in that case. Hence the performance comparison is not completely fair as the LEAP-based retroactive weaver detects the intended leak and 28 other objects that are not reclaimed and the load-time weaver does not. On the other hand, because the TOD system records events by augmenting the original program's bytecode, it is also not able to record events that occur in the code Java classes, and hence retroactively weaving the leak detector aspect on a TOD recording produces the same, less complete results as the load-time weaver. However, the aspect also depends on garbage collection for completeness. The prototype implementation of garbage collection in the TOD-based `VMM` is based on a naive mark-and-sweep algorithm implemented at the metaprogramming level, and is un-

surprisingly very slow. By contrast the LEAP-based `VMM` is based on a live JVM and can defer to its highly-optimized and asynchronous garbage collection.

Note that the RacerJ aspect cannot currently be used with the TOD-based backend because the TOD recorder and database does not record synchronization events and hence does not support the `lock()` and `unlock()` pointcuts. It is likely feasible to modify the TOD architecture to add this event type, however. An alternative scenario that only considers synchronized methods is possible, but ignoring the many synchronized blocks commonly used in concurrent code would likely produce many false positives.

It is important to put the runtime of retroactive weaving in perspective. The retroactive weaver is not used in the same way as the load-time weaver, and not intended to be an alternative to it nor competitive with it in runtime. The overhead reported above is somewhat comparable to the overhead of running a program in debug mode, and there is plenty of room for further optimization in future implementations. The advantage of time-travel programming is allowing the resources given to and context of a program's execution to vary independently from that of the execution analysis. Retroactive execution does not require human interaction to proceed, and is an ideal candidate for scheduling on idle computing resources. In addition, multiple analysis aspects can be woven retroactively against the same execution recording in parallel. Therefore, even though executing an aspect retroactively may be an order of magnitude slower that live execution, the ability to do so also saves the time otherwise needed to implement or reimplement the desired analysis at the metaprogramming level or within an execution query language.

6.8 Related Work

Several query languages have been proposed that treat program execution events as data, and define domain-specific constructs for selecting and aggregating that data. The Program Trace Query Language [28] is a SQL-like relational query language for Java based on several Java-specific relations for events such as object allocation and method invocation. Our work is complimentary to such query languages, which can extract values from multiple points in time at once, but cannot use the original source language to interpret those values. By contrast the Program

Query Language [42] uses flow-sensitive boolean conditions to identify application errors. PQL queries can be evaluated dynamically to catch errors at runtime, but can also be used to find sound approximations statically. Since the dynamic instrumentation application can invoke or replace application code, it constitutes a poincuts and advice language, one with more sophisticated pointcuts than AspectJ. The PQL implementation supports identifying query matches post-hoc, which is to say identifying matching join points given a program event stream, but does not support executing the bodies of queries.

The execution metaprogramming approach described earlier is closely related to the scriptable debugging paradigm, which involves driving debugging activities via a secondary language that operates on the debugging API as a first-class datatype. The secondary language may actually be a subset of the target language, as is the case with the Eclipse MAT utilities, but in that case the execution values of the debugged program are represented at the metaprogramming level and not as normal values within the debugging script or program. The Gnu debugger (GDB) features a Python scripting interface for automating debugging, and the recent Expositor [53] language supports time-travel debugging by allowing scripts to use traces as first-class objects, including moving backwards and forwards in time arbitrarily and retrieving, for example, the set of all points in time where a breakpoint would have been hit. Note that our mirror-based implementation of AspectJ only refers to the generic VM mirror interfaces, and do not require a holographic VM. If applied to an adapted JDI mirror, the process will dynamically load the aspect classes into the live process and execute advice bodies directly. This therefore can be seen as another scriptable debugging implementation[1] on a normal live process based on AOP.

Ansaloni et. al. [6] propose optimizing non-interfering aspects by executing them in parallel on multi-core architectures. To ensure consistency, however, the aspects in question have to be rewritten to ensure that the necessary state is explicitly copied into the aspect space so that base execution can continue without blocking. A possible improvement on this approach would be to use holographic

[1] Note that the completeness of this AspectJ implementation is limited by the JDI implementation; in particular general around advice is not supported for the version of JDI we used as its support for method invocation was not reentrant.

execution (in read-only mode) in combination with a copy-on-write layer to allow parallel execution of unaltered aspects. Their primary example of a non-interfering aspect is the Racer [21] algorithm that we included as a case study, which is compelling since it is a useful but costly aspect.

6.9 Summary

In this chapter we have shown that combining a reflective aspect weaver for AspectJ with the holographic JVM implementation from Chapter 5 creates an AspectJ retroactive weaver. We have shown that such a weaver can be used to evaluate a wide variety of aspects against execution recordings instead of inline with the original base program, with a high but not prohibitive runtime cost. We have also shown that adapting aspects to be free of illegal side-effects in their control flow only requires a handful of common auxiliary aspects and metaprogramming workarounds. While these results are very preliminary, they demonstrate the basic feasibility of this approach and lay the foundation for future implementations.

Chapter 7

Conclusion

We conclude with a review of the contributions of this thesis and how they support our thesis statement. We also outline the limitations and threats to validity of this work, and suggest several potential avenues of future research.

7.1 Summary

This thesis has presented Time-travel Programming, a novel programming paradigm for interacting with prior executions of software. Its defining characteristic is simulating the behaviour code would have produced if evaluated during such past executions. Its building blocks are Retroactive Execution, the simulation of evaluating code at some past time and context, and Retroactive Weaving, the retroactive execution of a post-hoc aspect across an execution recording. Our thesis statement is that TTP is effective, feasible, and flexible, which we have demonstrated as follows below.

TTP is effective: In Section 3.1, Section 5.3.1, and Section 6.6, we have outlined several examples of program analysis that are straightforward and logical to express using TTP, including debugging a long-standing memory leak in Eclipse.

TTP is feasible: By outlining a prototype for both retroactive execution and retroactive weaving for the Java Virtual Machine and the AspectJ AOP programming language in Chapter 5 and Chapter 6, we have provided evidence that TTP can be implemented for a full-featured programming language. We have shown

that the implementation effort required is reasonable by reusing substantial amounts of existing programming language implementation, and that it is feasible to work around missing information in execution recordings through moderate adaption effort of the post-hoc code.

TTP is flexible: Chapter 4 examined the concepts behind TTP in the context of a simple core language, and demonstrates how they interact with fundamental programming language concepts. This provides evidence that TTP is applicable to many programming languages that are built on these concepts. Orthogonally, by evaluating our prototype TTP implementation against three different sources of execution state (JVM heap dumps, execution traces and DR processes), we have demonstrated that the execution recording mechanism may vary independently from the TTP implementation for a particular programming language.

7.2 Limitations

Incomplete implementation of AspectJ: The reflective AspectJ weaver does not yet support the complete AspectJ language, although supporting the evaluation required a much more complete implementation that initially anticipated. Most omissions are from a lack of need for evaluation rather than any architectural obstacles.

Lack of core class weaving in `ajc`: For the most part the reflective AspectJ weaver supports join points that occur in the code of all classes, but the `ajc` toolset assumes an implementation based on bytecode rewriting and hence excludes some packages[1] from weaving. This artificially restricts valid aspects: advice on core methods only produce a compile warning, but intertype definitions on core classes produce compile errors even if load-time weaving is targeted. This means it is not possible to use `ajc` to produce the binary form (see Section 2.5) used by retroactive weaving of some valid retroactive aspects, which means developers must avoid AspectJ features that would otherwise provide the most elegant expression of their analysis.

[1] `java.*`, `javax.*`, `sun.reflect.*`, and `org.aspectj.*`. The latter package is excluded to avoid infinite recursion if the classes involved in the weaving implementation itself are woven.

7.3 Threats to Validity

Reflection may introduce unsound results: It is possible for the rich reflection interface in AspectJ to observe changes to type definitions or control flow, and this potential source of retroactive side-effects is not currently guarded against by the holographic JVM. This is unlikely to be a serious concern since the AspectJ language does not specify precisely how weaving affects these features of the runtime. For example, weaving often introduces extra stack frames between the caller and callee frames, and the exact nature of these changes depends on internal optimization decisions. Therefore, any code that depends on this data would be extremely fragile.

Double weaving of joint points in aspects: When `ajc` compiles aspects into their binary class form, it also applies compile-time weaving to the join points in those aspects. When such aspects are then loaded and woven by the reflective weaver against all events that occur in the runtime, advice may be applied twice to those join points. For the case studies in this thesis the only advice that applies to other advice code involves `cflow` tracking, whose application happens to be idempotent and hence does not affect execution semantics. Unfortunately, `ajc` does not expose any options for separating weaving from compiling, although it would be a relatively small change to support this in the tool's architecture.

Lack of generalization to other language implementations: The abstract concepts defined by TTP theoretically apply to any programming language. The general model described in Section 4.5 for retroactive weaving, for example, can theoretically be applied to many AOP languages, and provides a definition of the semantics of a retroactive weaver for any such language. However, the actual implementation techniques presented here may not be applicable for other AOP language runtimes, in particular for fundamentally different joinpoint models. The fact that our implementations overlapped significantly with existing language implementations is promising, but is no guarantee that other languages runtimes would be equally compatible with the TTP paradigm.

7.4 Future Work

Although we have shown that the implementation of our architecture is performant enough to be useful, there is still plenty of room for optimization that would greatly improve the scope of scenarios these techniques are feasible for. An important direction to pursue is to improve the bytecode loading and translating workflow in the holographic JVM implementation. Ideally, it should be possible to produce hologram bytecode once for each version of a class as it is developed, rather than repeatedly for each heap dump it occurs in. The translation process cannot currently be applied to a single class file in isolation, however, as the standard JVM bytecode type inferencing algorithm requires access to the context of the class hierarchy. Additional engineering effort should make it possible to remove this dependency, possibly by abstracting the type inferencing to leave placeholder types and replacing them with actual types only after caching the translated bytecode.

With regard to inferring class initialization state in the holographic JVM, Kozen and Stillerman [35] use a static analysis of class initializers similar to ours to initialize classes eagerly, in order to improve startup performance and catch errors earlier. Their algorithm ignores initializer effects with respect to static fields, but is flow-sensitive and hence calculates a more precise definition of initialization dependencies than our current implementation. Integrating their approach in the future may improve the success rate of our algorithm and hence reduce the number of initializers that must be marked safely repeatable.

It should be possible to apply the general model defined in Section 4.5 to AOP languages beyond those based on pointcuts and advice, such as crosscutting contracts as in AspectJML [57]. There are also several more radical paths other than retroactive weaving to explore in the intersection of AOP and execution recordings. It could be useful to expose future knowledge about the original execution through specialized pointcuts valid only in retroactive weaving; a `garbage(x)` pointcut, for example, that only matches join points where `x` is an object value that is garbage collectable at the end of execution. Such pointcuts would need to be designed carefully to only bind and clone immutable, self-contained values from the future environment, since retroactive execution of advice bodies cannot sensibly proceed in the context of more than one environment at once.

Similarly, given that execution recordings are often very large, it is intriguing to consider how time-travel programming can be parallelized without entirely losing the benefits of source code compatibility. For example, the workload of applying `toString` to every object in a heap dump as described in Section 5.3.2 could be parallelized by splitting the set of objects into multiple subsets, under the assumption that the effects of invoking `toString` on some objects do not affect the results of others. The portability of execution recordings and of holographic JVMs means that all state can be safely replicated in a distributed farm. Retroactive weaving of certain stateless or mostly stateless aspects could then be parallelized by partitioning the timeline of execution.

Finally, this work to date has focussed exclusively on analyzing a single process at once, but computer systems are becoming increasingly multi-process and distributed. The scope of time-travel programming as a concept could easily be extended to evaluate additional code in the context of multiple independently recorded processes. This could be applied very naturally to actor-based systems, where retroactive aspects in the applicable AOP language could create additional actors themselves in order to communicate analysis data across processes, again restoring the advantages of a programming paradigm even when that programming happens after the fact.

Bibliography

[1] Clozure cl documentation - 4.9. saving applications. URL http://ccl.clozure.com/manual/chapter4.9.html#Saving-Applications. Accessed October 2013. → pages 79

[2] The AspectJ programming guide. URL https://eclipse.org/aspectj/doc/released/progguide/index.html. Accessed May 2017. → pages 14

[3] Eclipse memory analyzer open source project. URL http://www.eclipse.org/mat/. Accessed October 2013. → pages 20

[4] VAssert programming guide, 2008. URL https://www.vmware.com/pdf/ws65_vassert_programming.pdf. Accessed June 2011. → pages 24, 28

[5] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005. → pages 79

[6] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 1–12, New York, NY, USA, 2010. ACM. → pages 96, 104

[7] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320. USENIX Association, 2012. → pages 26

[8] T. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In *Proceedings of the 2011 ACM international conference on Object*

oriented programming systems languages and applications, pages 921–938. ACM, 2011. → pages 78

[9] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 86–95, New York, NY, USA, 2002. ACM. → pages 16

[10] S. Balzer, P. T. Eugster, and B. Meyer. Can Aspects Implement Contracts? In N. Guelfi and A. Savidis, editors, *Rapid Integration of Software Engineering Techniques*, number 3943 in Lecture Notes in Computer Science, pages 145–157. Springer Berlin Heidelberg, 2006. → pages 93

[11] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005. → pages 32

[12] E. Bodden and K. Havelund. Racer: effective race detection using aspectj. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 155–166, New York, NY, USA, 2008. ACM. ACM ID: 1390650. → pages 25, 96

[13] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 331–344. ACM, 2004. → pages 59, 79

[14] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002. → pages 64

[15] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. → pages 68

[16] K. Chen and J.-B. Chen. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 23–28, July 2007. → pages 97

[17] J. Choi and B. Alpern. DejaVu: Deterministic Java Replay Debugger for Jalapeno Java Virtual Machine. *OOPSLA 2000 Companion*, 2000. → pages 26

[18] J. D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001. → pages 2

[19] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, 2008. → pages 35

[20] J. Cook and A. Nusayr. Using AOP for Detailed Runtime Monitoring Instrumentation. In *WODA 2008: the sixth international workshop on dynamic analysis*. → pages 25

[21] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252. ACM, 1977. → pages 70

[22] D. S. Dantas and D. Walker. Harmless advice. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 383–396, New York, NY, USA, 2006. ACM. → pages 17

[23] B. De Fraine, E. Ernst, and M. Südholt. Essential AOP: The A Calculus. In *ECOOP 2010 - Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 101–125. Springer Berlin Heidelberg, 2010. → pages 53

[24] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 525–540. USENIX Association, 2014. → pages 18, 26

[25] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *Operating Systems Design and Implementation*, 2002. → pages 18, 34, 35

[26] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63 (3), Dec. 2006. ISSN 0167-6423. → pages 16, 40, 53

[27] M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, page 288300. ACM, 2004. → pages 65

[28] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages, and Applications*, page 402, 2005. → pages 24, 103

[29] O. Gruber, B. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Systems Journal*, 44(2): 289–299, 2005. → pages 22

[30] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM. → pages 15

[31] J. Huang, P. Liu, and C. Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 207–216, 2010. → pages 18, 91

[32] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM. → pages 28

[33] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586. → pages 10

[34] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. → pages 40

[35] D. Kozen and M. Stillerman. Eager class initialization for java. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2*, FTRTFT '02, pages 71–80. Springer-Verlag, 2002. → pages 80, 109

[36] S. Krishnamurthi. *Programming Languages: Application and Interpretation*. 2007. URL http://www.plai.org. Accessed March 2015. → pages 38

[37] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. In *EuroSys*, 2009. → pages 32

[38] B. Lewis. Debugging backwards in time. In *Automated and Analysis-Driven Debugging*, 2003. → pages 18

[39] K. J. Lieberherr and D. Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press. → pages 45

[40] D. H. Lorenz and J. Vlissides. Pluggable reflection: decoupling meta-interface and implementation. In *25th International Conference on Software Engineering, 2003. Proceedings*, pages 3– 13. IEEE, May 2003. → pages 79

[41] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, 2005. → pages 24

[42] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. *Object-Oriented Programming, Systems, Languages and Applications*, 2005. → pages 104

[43] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP 2003 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28. Springer Berlin Heidelberg, 2003. → pages 13, 45

[44] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 115–124. ACM, 2010. → pages 80

[45] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, Oct. 1992. → pages 93

[46] S. Mostinckx, T. Van Cutsem, S. Timbermont, and E. Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 89–100. ACM, 2007. → pages 78

[47] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*. 2002. → pages 29

[48] T. Ngo and J. Barton. Debugging by remote reflection. In *Euro-Par 2000 Parallel Processing*, pages 1031–1038, 2000. → pages 26

[49] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Architectural Support for Programming Languages and Operating Systems*, 2008. → pages 2, 26

[50] Oracle. Java platform debugger architecture. URL http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/index.html. Accessed October 2013. → pages 59

[51] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7, New York, NY, USA, May 2005. ACM. ACM ID: 1083251. → pages 2

[52] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, 2007. → pages 98

[53] K. Y. Phang, J. S. Foster, and M. Hicks. EXPOSITOR: Scriptable Time-Travel Debugging with First Class Traces. In *Proceedings of the 2013 International Conference on Software Engineering*, 2013. → pages 104

[54] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Aspect-Oriented Software Development*, 2003. → pages 16

[55] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 558–582, Berlin, Heidelberg, 2011. Springer-Verlag. → pages 18, 47, 89

[56] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. *ACM SIGPLAN Notices*, 42(10), 2007. → pages 18, 89, 90

[57] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm. AspectJML: Modular specification and runtime checking for crosscutting contracts. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 157–168, New York, NY, USA, 2014. ACM. → pages 93, 109

[58] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, Jan. 1965. ISSN 0004-5411. → pages 30

[59] R. Salkeld and R. Garcia. Essential Retroactive Weaving. In *Companion Proceedings of the 14th International Conference on Modularity*, MODULARITY Companion 2015, pages 52–57, New York, NY, USA, 2015. ACM. → pages v

[60] R. Salkeld and G. Kiczales. Interacting with dead objects. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 203–216, New York, NY, USA, 2013. ACM. → pages v, 49

[61] R. Salkeld, W. Xu, B. Cully, G. Lefebvre, A. Warfield, and G. Kiczales. Retroactive aspects: Programming in the past. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA '11, pages 29–34, New York, NY, USA, 2011. ACM. → pages v

[62] V. Schuppan, M. Baur, and A. Biere. JVM independent replay in java. *Electronic Notes in Theoretical Computer Science*, 113:85–104, Jan. 2005. → pages 2

[63] M. X. Sheldon, G. V. Weissman, and V. M. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Modeling, Benchmarking and Simulation*, 2007. → pages 2, 35

[64] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 23–35, New York, NY, USA, 1984. ACM. → pages 10

[65] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, Orlando, Florida, United States, 1994. ACM. → pages 24

[66] A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 159–167, New York, NY, USA, 2008. ACM. → pages 97, 98, 102

[67] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 85–94. ACM, 2007. → pages 79

[68] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: object centric deterministic replay for java. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, page 3030. USENIX Association, 2011. → pages 18

Appendix A

Appendices

A.1 Illegal Native Methods in the JRE

This appendix contains our classification of all illegal native methods in the JRE. See Table A.1 for the complete list.

Category	Class	Method
Core Class Initialization	*	registerNatives
	java.lang.System	initProperties
	java.util.concurrent.atomic.AtomicLong	VMSupportsCS8
	sun.misc.VM	initialize
Drivers	sun.print.*	*
	sun.security.smartcardio.*	*
GUI	apple.laf.*	*
	com.apple.eawt.*	*
	com.apple.laf.*	*
	com.sun.java.swing.plaf.gtk.*	*
	java.awt.*, sun.awt.*	*
	sun.lwawt.*	*
Graphics	com.sun.imageio.plugins.jpeg.*	*
	sun.dc.pr.*	*
	sun.font.*	*
	sun.java2d.*	*

Category	Class	Method
IO	com.apple.eio.*	*
	com.sun.java.util.jar.pack.NativeUnpack	*
	java.io.*, java.nio.*, sun.nio.*	*
	java.util.logging.FileHandler	*
	java.util.prefs.*	*
	sun.misc.MessageUtils	*
JIT Compilation	java.lang.Compiler	*
Java 7 Method Handles	java.lang.invoke.*	*
Management	com.sun.demo.jvmti.hprof.*	*
	oracle.jrockit.jfr.*	*
	sun.management.*	*
	sun.misc.Perf	*
	sun.tracing.dtrace.JVM	*
Media	com.sun.media.sound.*	*
Native Libraries	java.lang.ClassLoader$NativeLibrary	*
	java.lang.System	mapLibraryName
Network	java.net.*, sun.net.*	*
	sun.rmi.*	*
Security	apple.security.*, sun.security.*	*

Category	Class	Method
Shared superclass	`java.lang.Object`	`*`
	`java.lang.Throwable`	`*`
System	`apple.applescript.*`	`*`
	`apple.launcher.*`	`*`
	`com.apple.concurrent.*`	`*`
	`com.apple.jobjc.*`	`*`
	`com.apple.resources.LoadNativeBundleAction`	`*`
	`com.sun.management.*`	`*`
	`java.lang.ProcessEnvironment`	`environ`
	`java.lang.Runtime`	`*`
	`java.lang.UNIXProcess`	`*`
	`java.util.TimeZone`	`*`
	`sun.misc.GC`	`*`
	`sun.misc.NativeSignalHandler`	`*`
	`sun.misc.Signal`	`*`

Table A.1: Categorization of forbidden methods in the Java Runtime Environment

A.2 Illegal Side-effects in AspectJ Case Studies

This appendix contains the complete list of illegal side-effects encountered in the control flow of the retroactive aspects evaluated in Chapter 6. Table A.2 summarizes the state that the execution the aspects attempted to modify, and how these effects were elided with either additional aspects or (in the case of effects encountered when booting the holographic JVM) low-level metaprogramming. The second column indicates whether the side-effects occurred when loading aspects themselves, making it necessary to implement the strategy using metaprogramming instead. The strategies named in the third column are as follows:

Skip method: Simply removing invocations of methods that are not necessary for correctness.

Avoiding cache: Eliding the side-effect, nullifying the effect of the relevant caching.

Secondary cache: Duplicating the referenced storage so that retroactive execution reads and writes independently but equivalently.

Secondary counter: Duplicating a numerical counter that is incremented to generate distinct values. This strategy is valid as long as overlapping retroactive values with original values does not interfere with soundness.

Augmenting collection: Allocating a secondary collection of values that semantically augments the original and advising interactions with that collection to maintain consistency, as illustrated in Figure 6.3.

Secondary I/O: Replacing standard output streams with independent streams that can be explicitly accessed by TTP clients.

Reduced concurrency: A specific case during class loading to avoid updating storage that tracks locks per class, wherein the class loader itself is synchronized on instead. This leads to decreased concurrency in retroactive weaving but does not interfere with correctness.

Table A.2: Illegal side-effects encountered by case studies

State	Bootstrap	Strategy
java.io.UnixFileSystem.cache	y	Secondary cache
java.io.UnixFileSystem.javaHomePrefixCache	y	Secondary cache
java.lang.ApplicationShutdownHooks.hooks		Augmented collection
java.lang.Class.declaredFields	y	Secondary cache
java.lang.ClassLoader.getClassLoadingLock	y	Reduced concurrency
java.lang.String.hash	y	Avoided cache
java.lang.System.stdout		Secondary I/O
java.lang.System.stderr		Secondary I/O
java.lang.Thread.threadInitNumber	y	Secondary counter
java.lang.Thread.threadSeqNumber	y	Secondary counter
java.lang.ThreadLocal.nextHashCode	y	Secondary counter
java.lang.ThreadLocal values	y	Augmented collection
java.net.URLClassLoader.closeables	y	Augmented collection
java.nio.charset.Charset.cache1	y	Secondary cache
java.nio.charset.Charset.cache2	y	Secondary cache
java.util.zip.ZipCoder.dec	y	Secondary cache
java.util.zip.ZipCoder.enc	y	Secondary cache
sun.reflect.NativeConstructorAccessorImpl.numInvocations	y	Secondary counter
sun.misc.Hashing.randomHashSeed	y	Secondary counter
sun.misc.MetaIndex.registerDirectory	y	Skip method
sun.nio.cs.ThreadLocalCoders$Cache.cache	y	Secondary cache

A.3 RAPL Interpreter Source Code

This appendix contains the complete source code for the RAPL interpreter described in Chapter 4.

Listing A.1: Complete RAPL interpreter source code

```
#lang plai

(require "rapl.rkt")
(require "rapl_parser.rkt")
(require "rapl_serialization.rkt")

;;
;; Rapl interpreter
;;

(define-type Value
  (numV (n number?))
  (boolV (b boolean?))
  (symbolV (s symbol?))
  (closV (params (listof symbol?)) (body ExprC?) (env Env?))
  (boxV (l Location?))
  (voidV)
  (taggedV (tag Value?) (value Value?))
  (traceValueV (v Value?))
  (resumeV (label string?) (pos number?)))

(define-type Binding
  [bind (name symbol?) (value Value?)])
(define Location? number?)
(define Env? (listof Binding?))
(define mt-env empty)

(define-type Storage
  [cell (location Location?) (val Value?)])
```

```
(define Store? (listof Storage?))
(define mt-store empty)

(define-type Advice
  [aroundappsA (advice Value?)])
(define AdvStack? (listof Advice?))
(define mt-adv empty)

(define-type Control
  [interp-init]
  [app-call (abs Value?) (args (listof Value?))]
  [app-result (r Value?)])

(define-type State
  [state (c Control?) (adv AdvStack?) (sto Store?) (tin TraceIn?)])

(define-type TraceOut
  [traceout (states (listof State?))])
(define mt-traceout (traceout empty))

(define/contract (append-traceout . ts) (->* () () #:rest (listof TraceOut?) TraceOut?)
  (traceout (foldl append '() (map traceout-states (reverse ts)))))

(define-type TraceIn
  [tracein (states (listof State?))])
(define mt-tracein (tracein empty))

(define/contract (trace-state tin) (-> TraceIn? State?)
  (first (tracein-states tin)))

(define/contract (next-trace-state tin) (-> TraceIn? TraceIn?)
```

```
  (tracein (rest (tracein-states tin))))

(define-type Result
  [v*s*t*t (v Value?) (s Store?) (tin TraceIn?) (tout TraceOut?)])

;; Numbers and arithmetic

(define (num+ l r)
  (cond
    [(and (numV? l) (numV? r))
     (numV (+ (numV-n l) (numV-n r)))]
    [else
     (error 'num+ "one argument was not a number")]))

(define (num* l r)
  (cond
    [(and (numV? l) (numV? r))
     (numV (* (numV-n l) (numV-n r)))]
    [else
     (error 'num* "one argument was not a number")]))

(define (numWrite v)
   (cond
    [(numV? v)
     (write (numV-n v))]
    [else
     (error 'numWrite "argument was not a number")]))

;; Booleans and conditionals

(define/contract (deep-untag v) (-> Value? Value?)
```

```
  (type-case Value v
    [taggedV (tag tagged)
             (deep-untag tagged)]
    [else v]))

(define/contract (equal-values l r) (-> Value? Value? boolean?)
  (equal? l r))

;; Identifiers and functions

(define/contract (lookup for env) (-> symbol? Env? Value?)
  (cond
    [(empty? env) (error 'lookup (string-append "name not found: " (symbol->string for)))]
    [else
     (type-case Binding (first env)
       [bind (name value)
             (cond
               [(symbol=? for name) value]
               [else (lookup for (rest env))])])]))

(define/contract (apply f args adv sto tin) (-> Value? (listof Value?) AdvStack? Store? TraceIn? Result?)
  (type-case Value (deep-untag f)
    [closV (params body env)
           (let ([bs (map bind params args)])
             (interp body (append bs env) adv sto tin))]
    [resumeV (label pos)
             (if (unbox retroactive-error-checking)
                 (rw-call pos args adv sto tin)
                 (rw-call-no-error args adv sto tin))]
    [traceValueV (tf)
                 (apply (lift-trace-value tf) args adv sto tin)]
```

```
    [else (error (string-append "only functions can be applied: " (value->string f)))]))

(define z-combinator
  (parse-string "(lambda (f) ((lambda (x) (f (lambda (y) ((x x) y))))
                              (lambda (x) (f (lambda (y) ((x x) y)))))))"))

;; Mutations and side-effects

(define/contract (storage-at storage loc) (-> (listof Storage?) Location? (or/c Storage? #f))
  (cond
    [(empty? storage) #f]
    [else
     (let ([s (first storage)])
       (type-case Storage s
         [cell (l val)
               (cond
                 [(= loc l) s]
                 [else (storage-at (rest storage) loc)])]))]))

(define/contract (fetch sto tin b) (-> Store? TraceIn? Value? Value?)
  (type-case Value (deep-untag b)
    [boxV (loc)
          (let ([storage (storage-at sto loc)])
            (if storage
                (type-case Storage storage
                  [cell (l val) val])
                (error "location not found")))]
    [traceValueV (v)
                 (type-case State (trace-state tin)
                   [state (c adv sto-t tin-t)
                          (fetch sto-t tin-t v)])]
```

```
    [else (error "attempt to unbox a non-box")]))

(define/contract (new-loc sto) (-> Store? Location?)
 (length sto))

(define override-store cons)

(define (list-box-push! b x)
  (set-box! b (cons x (unbox b))))
(define (list-box-pop! b)
  (let* ([next (first (unbox b))]
         [_ (set-box! b (rest (unbox b)))])
         next))

(define read-source (box (lambda (prompt)
                           (display prompt)
                           (display "> ")
                           (string->number (read-line)))))
(define write-sink (box (lambda (s) (begin (display s) (newline)))))

;; Lifting trace values

(define/contract (lift-binding b) (-> Binding? Binding?)
  (type-case Binding b
    [bind (name value)
          (bind name (lift-trace-value value))]))

(define/contract (lift-trace-value v) (-> Value? Value?)
  (type-case Value v
    [numV (_) v]
    [boolV (_) v]
```

```
    [symbolV (_) v]
    [closV (params body env)
           (closV params body (map lift-binding env))]
    [boxV (l) (traceValueV v)]
    [traceValueV (tv) (traceValueV v)]
    [voidV () v]
    [taggedV (tag tagged)
             (taggedV (lift-trace-value tag) (lift-trace-value tagged))]
    [resumeV (label pos) v]))

(define/contract (prepend-trace t r) (-> TraceOut? Result? Result?)
  (type-case Result r
    [v*s*t*t (v-r s-r tin-r tout-r)
             (v*s*t*t v-r s-r tin-r (append-traceout t tout-r))]))

;; Advice

(define/contract (apply-with-weaving f args adv adv2 sto tin) (-> Value? (listof Value?) AdvStack?
    AdvStack? Store? TraceIn? Result?)
  (type-case Result (weave adv adv2 f sto tin)
      (v*s*t*t (woven-f s-w tin-w tout-w)
               (type-case Result (apply woven-f args adv2 s-w tin-w)
                 (v*s*t*t (r s-r tin-r tout-r)
                          (let ([call-state (state (app-call f args) adv sto tin)]
                                [return-state (state (app-result r) adv s-r tin-r)])
                            (v*s*t*t r s-r tin-r (append-traceout (traceout (list call-state))
                                                                  tout-w
                                                                  tout-r
                                                                  (traceout (list return-state)))))))))

; Applies all advice in scope for all tags on f
```

```
(define/contract (weave adv adv2 f sto tin) (-> AdvStack? AdvStack? Value? Store? TraceIn? Result?)
  (type-case Value f
    [taggedV (tag tagged)
             (type-case Result (weave adv adv2 tagged sto tin)
               [v*s*t*t (v-w s-w tin-w tout-w)
                        (prepend-trace tout-w (weave-for-tag adv adv2 tag v-w s-w tin-w))])]
    [else (v*s*t*t f sto tin mt-traceout)]))

; Applies all advice in scope for a single tag on f
(define/contract (weave-for-tag adv adv2 tag f sto tin) (-> AdvStack? AdvStack? Value? Value? Store?
    TraceIn? Result?)
  (if (empty? adv)
      (v*s*t*t f sto tin mt-traceout)
      (type-case Result (weave-advice adv2 tag (first adv) f sto tin)
        [v*s*t*t (v-w s-w tin-w tout-w)
                 (prepend-trace tout-w (weave-for-tag (rest adv) adv2 tag v-w s-w tin-w))])))

; Apply a single advice function to f
(define/contract (weave-advice adv tag advice f sto tin) (-> AdvStack? Value? Advice? Value? Store?
    TraceIn? Result?)
  (type-case Advice advice
    [aroundappsA (g)
                 (apply g (list tag f) adv sto tin)]))

;; Debugging

(define/contract (display-value v out) (-> Value? output-port? void?)
  (type-case Value v
    [numV (n) (display n out)]
    [boolV (b) (display b out)]
    [symbolV (s) (write s out)]
```

```
244     [closV (params body env)
245            (begin (display params out) (display " -> " out) (display (exp-syntax body) out) (newline
                   out) (display-env env out))]
246     [boxV (l)
247           (begin (display "box(" out) (display l out) (display ")" out))]
248     [traceValueV (v)
249           (begin (display "tracevalue(" out) (display-value v out) (display ")" out))]
250     [voidV ()
251            (display "(void)" out)]
252     [taggedV (t v)
253             (begin (display "(tag " out) (display-value t out) (display " " out) (display-value v out)
                    (display ")" out))]
254     [resumeV (label f) (display label out)]))
255
256 (define/contract (value->string v) (-> Value? string?)
257   (letrec ([out (open-output-string)]
258            [_ (display-value v out)])
259     (get-output-string out)))
260
261 (define/contract (display-context env sto t out) (-> Env? Store? TraceIn? output-port? void?)
262   (begin (display "=======================================\n" out)
263          (display "Environment: \n" out)
264          (display-env env out)
265          (display "Store: \n" out)
266          (display-store sto out)
267          (if (empty? (tracein-states t))
268              (void)
269              (begin (display "Trace Store: \n" out)
270                     (display-store (state-sto (trace-state t)) out)))))
271
272 (define/contract (display-env env out) (-> Env? output-port? void?)
```

```
  (for ([def env])
    (type-case Binding def
      [bind (n v)
            (begin (display "\t\t" out) (display n out) (display " -> " out) (display-value v out)
                (display "\n" out))])))

(define/contract (display-store sto out) (-> Store? output-port? void?)
  (for ([c sto])
    (type-case Storage c
      [cell (l v)
            (begin (display "\t" out) (display l out) (display " -> " out) (display-value v out)
                (display "\n" out))])))

(define/contract (display-state s out) (-> State? output-port? void?)
  (type-case State s
    [state (c adv sto tin)
           (type-case Control c
             [interp-init ()
                          (begin (display "(interp-init)" out))]
             [app-call (f args)
                          (begin (display "(app-call " out) (display-value f out) (display ")" out))]
             [app-result (result)
                           (begin (display "(app-return " out) (display-value result out) (display ")"
                               out))])]))

(define/contract (display-advice a out) (-> Advice? output-port? void?)
  (display-value (aroundappsA-advice a) out))

(define/contract (display-advice-stack adv out) (-> AdvStack? output-port? void?)
  (begin (display "[" out) (display "\n" out)
         (map (lambda (a) (begin (display " " out) (display-advice a out) (display "," out) (display
```

```
            "\n" out))) adv)
301         (display "]")))
302
303 (define/contract (display-with-label label val out) (-> string? Value? output-port? void?)
304   (begin (display label out) (display ": " out) (display-value val out) (newline out)))
305
306 ;; Main interpretation function
307
308 (define verbose-interp (box false))
309 (define retroactive-error-checking (box true))
310
311 (define/contract (interp expr env adv sto tin) (-> ExprC? Env? AdvStack? Store? TraceIn? Result?)
312 (begin
313   (if (unbox verbose-interp)
314       (begin
315         (display "Expression: ") (display (exp-syntax expr)) (newline)
316         (display-context env sto tin (current-output-port))
317         (newline))
318       '())
319
320   (type-case ExprC expr
321
322     ;; Numbers and arithmetic
323
324     [numC (n) (v*s*t*t (numV n) sto tin mt-traceout)]
325
326     [plusC (l r) (type-case Result (interp l env adv sto tin)
327                 [v*s*t*t (v-l s-l tin-l tout-l)
328                          (type-case Result (interp r env adv s-l tin-l)
329                            [v*s*t*t (v-r s-r tin-r tout-r)
330                                     (v*s*t*t (num+ (deep-untag v-l) (deep-untag v-r)) s-r tin-r
```

```
                                        (append-traceout tout-l tout-r))])])]

    [multC (l r) (type-case Result (interp l env adv sto tin)
                   [v*s*t*t (v-l s-l tin-l tout-l)
                            (type-case Result (interp r env adv s-l tin-l)
                              [v*s*t*t (v-r s-r tin-r tout-r)
                                       (v*s*t*t (num* (deep-untag v-l) (deep-untag v-r)) s-r tin-r
                                                (append-traceout tout-l tout-r))])])]

    ;; Booleans and conditionals

    [boolC (b) (v*s*t*t (boolV b) sto tin mt-traceout)]

    [equalC (l r) (type-case Result (interp l env adv sto tin)
                    [v*s*t*t (v-l s-l tin-l tout-l)
                             (type-case Result (interp r env adv s-l tin-l)
                               [v*s*t*t (v-r s-r tin-r tout-r)
                                        (v*s*t*t (boolV (equal-values (deep-untag v-l) (deep-untag
                                            v-r))) s-r tin-r
                                             (append-traceout tout-l tout-r))])])]

    [ifC (c t f) (type-case Result (interp c env adv sto tin)
                   [v*s*t*t (v-c s-c tin-c tout-c)
                            (type-case Result (if (boolV-b (deep-untag v-c))
                                                  (interp t env adv s-c tin-c)
                                                  (interp f env adv s-c tin-c))
                              [v*s*t*t (v-b s-b tin-b tout-b)
                                       (v*s*t*t v-b s-b tin-b (append-traceout tout-c tout-b))])])]

    ;; Identifiers and abstractions
```

```
361     [idC (n) (v*s*t*t (lookup n env) sto tin mt-traceout)]
362
363     [lamC (params b) (v*s*t*t (closV params b env) sto tin mt-traceout)]
364
365     [appC (f args) (type-case Result (interp f env adv sto tin)
366                      [v*s*t*t (v-f s-f tin-f tout-f)
367                               (type-case ResultList (map-expr-list (lambda (e s t) (interp e env adv s t))
                                                          args s-f tin-f)
368                                 [vs*s*t*t (v-args s-args tin-args tout-args)
369                                           (prepend-trace (append-traceout tout-f tout-args)
370                                                          (apply-with-weaving v-f v-args adv adv s-args
                                                              tin-args))])])]
371
372     [recC (f) (interp (appC z-combinator (list f)) env adv sto tin)]
373
374     [letC (s v in) (type-case Result (interp v env adv sto tin)
375                      [v*s*t*t (v-v s-v tin-v tout-v)
376                               (prepend-trace tout-v (interp in (cons (bind s v-v) env) adv s-v
                                                          tin-v))])]
377
378     ;; Boxes and sequencing
379
380     [boxC (a) (type-case Result (interp a env adv sto tin)
381                 [v*s*t*t (v-a s-a tin-a tout-a)
382                          (let ([where (new-loc sto)])
383                            (v*s*t*t (boxV where)
384                                     (override-store (cell where v-a) sto)
385                                     tin-a
386                                     tout-a))])]
387
388     [unboxC (a) (type-case Result (interp a env adv sto tin)
```

```
                [v*s*t*t (v-a s-a tin-a tout-a)
                         (v*s*t*t (fetch s-a tin-a v-a) s-a tin-a tout-a)])]

    [setboxC (b val) (type-case Result (interp b env adv sto tin)
                       [v*s*t*t (v-b s-b tin-b tout-b)
                                (type-case Result (interp val env adv s-b tin-b)
                                  [v*s*t*t (v-v s-v tin-v tout-v)
                                           (type-case Value (deep-untag v-b)
                                             [boxV (l) (v*s*t*t (voidV)
                                                               (override-store (cell l v-v) s-v)
                                                               tin-v
                                                               (append-traceout tout-b tout-v))]
                                             [traceValueV (v) (error 'retroactive-side-effect "attempt
                                                 to retroactively set box")]
                                             [else (error 'interp "attempt to set-box! on a
                                                 non-box")])])])]

    [seqC (b1 b2) (type-case Result (interp b1 env adv sto tin)
                    [v*s*t*t (v-b1 s-b1 tin-b1 tout-b1)
                             (prepend-trace tout-b1 (interp b2 env adv s-b1 tin-b1))])]

    [voidC () (v*s*t*t (voidV) sto tin mt-traceout)]

    ;; Advice

    [symbolC (s) (v*s*t*t (symbolV s) sto tin mt-traceout)]

    [tagC (tag v)
          (interp-tag tag v env adv sto tin)]

    [aroundappsC (advice extent)
```

```
                 (interp-aroundapps advice extent env adv sto tin)]

    ;; Input/Output

    [fileC (path) (interp (parse-file path) mt-env adv sto tin)]

    [writeC (l a) (type-case Result (interp a env adv sto tin)
                [v*s*t*t (v-a s-a tin-a tout-a)
                         (begin ((unbox write-sink) (string-append l ": " (value->string v-a)))
                                (v*s*t*t (voidV) s-a tin-a tout-a))])]

    [readC (l) (let* ([val ((unbox read-source) l)]
                      [_ (record-interp-input val)])
                 (v*s*t*t (numV val) sto tin mt-traceout))]))
)

(define/contract (interp-tag tag v env adv sto tin) (-> ExprC? ExprC? Env? AdvStack? Store? TraceIn?
    Result?)
  (type-case Result (interp tag env adv sto tin)
    [v*s*t*t (v-tag s-tag tin-tag tout-tag)
             (type-case Result (interp v env adv s-tag tin-tag)
               [v*s*t*t (v-v s-v tin-v tout-v)
                        (v*s*t*t (taggedV v-tag v-v) s-v tin-v (append-traceout tout-tag tout-v))])]))

(define/contract (interp-aroundapps advice extent env adv sto tin) (-> ExprC? ExprC? Env? AdvStack?
    Store? TraceIn? Result?)
  (type-case Result (interp advice env adv sto tin)
    [v*s*t*t (v-a s-a tin-a tout-a)
             (let ([new-adv (cons (aroundappsA v-a) adv)])
               (prepend-trace tout-a (interp extent env new-adv s-a tin-a)))]))
```

```
(define-type ResultList
  [vs*s*t*t (vs (listof Value?)) (s Store?) (tin TraceIn?) (tout TraceOut?)])
(define/contract (append-result rl r) (-> ResultList? Result? ResultList?)
  (type-case ResultList rl
    (vs*s*t*t (vs old-s old-tin old-tout)
              (type-case Result r
                (v*s*t*t (v s tin tout)
                         (vs*s*t*t (append vs (list v)) s tin (append-traceout old-tout tout)))))))

(define/contract (map-expr-list f exprs sto tin) (-> (-> ExprC? Store? TraceIn? Result?) (listof ExprC?)
    Store? TraceIn? ResultList?)
  (let ([helper (lambda (e rl)
                  (type-case ResultList rl
                    [vs*s*t*t (vs s tin-rl tout-rl)
                              (append-result rl (f e s tin-rl))]))])
    (foldl helper (vs*s*t*t '() sto tin mt-traceout) exprs)))

(define/contract (interp-exp exp) (-> ExprC? Value?)
  (v*s*t*t-v (interp exp mt-env mt-adv mt-store mt-tracein)))

(define/contract (app-chain exps) (-> (listof ExprC?) ExprC?)
  (foldl (lambda (next chained) (appC chained next)) (first exps) (rest exps)))

;; Replay

(define interp-input (box '()))

(define (record-interp-input (x number?))
  (list-box-push! interp-input x))
(define get-interp-input
  (lambda () (reverse (unbox interp-input))))
```

```
477
478 (define-type RaplRecording
479   [raplRecForReplay (program list?) (input list?)])
480
481 (define/contract (interp-with-recording exps recording-path) (-> (listof ExprC?) path-string? Result?)
482   (let* ([result (interp-exp (app-chain exps))]
483          [input (get-interp-input)]
484          [recording (raplRecForReplay exps input)]
485          [_ (write-struct-to-file recording recording-path)])
486     result))
487
488 (define/contract (replay-interp recording-path) (-> path-string? Result?)
489   (let* ([recording (read-struct-from-file recording-path)]
490          [remaining-input (box (raplRecForReplay-input recording))]
491          [_ (set-box! read-source (lambda (prompt) (list-box-pop! remaining-input)))])
492     ((interp-exp (app-chain (raplRecForReplay-program recording))))))
493
494 ;; Tracing
495
496 (define/contract (interp-with-tracing exprs trace-path) (-> (listof ExprC?) path-string? Value?)
497   (type-case Result (interp (app-chain exprs) mt-env mt-adv mt-store mt-tracein)
498     [v*s*t*t (v s tin tout)
499            (let ([trace (append (list (state (interp-init) mt-adv mt-store mt-tracein))
500                                 (traceout-states tout)
501                                 (list (state (app-result v) mt-adv s tin)))])
502              (begin
503                (if (file-exists? trace-path)
504                    (delete-file trace-path)
505                    (void))
506                (write-struct-to-file trace trace-path)
507                v))]))
```

```
508
509 ;; TODO-RS: Gah, can't figure out how to get a hold of the current module
510 (define rapl-ns (module->namespace (string->path
        "/Users/robinsalkeld/Documents/UBC/Code/rapl/rapl_interpreter.rkt")))
511
512 (define/contract (interp-query trace-path exprs) (-> path-string? (listof ExprC?) Value?)
513   (let* ([_ (set-box! read-source (lambda (prompt) (error 'retroactive-side-effect "attempt to
        retroactively read input")))]
514          [tin (tracein (read-struct-from-file rapl-ns trace-path))]
515          [resume (resumeV "top-level thunk" (length (tracein-states tin)))])
516     (type-case Result (interp (app-chain exprs) mt-env mt-adv mt-store mt-tracein)
517       [v*s*t*t (v-a s-a tin-a tout-a)
518                (type-case Result (apply-with-weaving v-a (list resume) mt-adv mt-adv s-a tin)
519                  [v*s*t*t (v-t s-t tin-t tout-t)
520                           (type-case Result (apply v-t (list) mt-adv s-t tin-t)
521                             [v*s*t*t (v-r s-r tin-r tout-r)
522                                      (if (or #t (= (length (tracein-states tin-r)) 0))
523                                          v-r
524                                          (error 'interp-query "Trace not fully read ~s" tin-r))])])])))
525
526 (define/contract (all-tags v) (-> Value? (listof Value?))
527   (type-case Value v
528     [taggedV (tag tagged)
529              (cons tag (all-tags tagged))]
530     [else empty]))
531
532 (define/contract (deep-tag tags v) (-> (listof Value?) Value? Value?)
533   (foldr taggedV v tags))
534
535 (define/contract (is-trace-advice? a) (-> Advice? boolean?)
536   (type-case Advice a
```

```
537     [aroundappsA (advice)
538                  (traceValueV? advice)]))
539
540 (define/contract (lift-advice a) (-> Advice? Advice?)
541   (type-case Advice a
542     [aroundappsA (advice)
543                  (aroundappsA (traceValueV advice))]))
544
545 (define/contract (new-trace-advice trace-adv adv) (-> AdvStack? AdvStack? AdvStack?)
546   (if (empty? adv)
547       (map lift-advice trace-adv)
548       (if (is-trace-advice? (first adv))
549           (new-trace-advice (rest trace-adv) (rest adv))
550           (new-trace-advice trace-adv (rest adv)))))
551
552 (define/contract (without-trace-advice adv) (-> AdvStack? AdvStack?)
553   (let ([result (filter (lambda (a) (not (is-trace-advice? a))) adv)])
554     (begin ;(display "with trace advice: ") (display-list adv) (newline)
555            ;(display "without trace advice: ") (display-list result) (newline)
556            result)))
557
558 (define (display-list l)
559   (begin (display "[") (newline)
560          (map (lambda (e) (begin (display " ") (display e) (display ",") (newline))) l)
561          (display "]")))
562
563 (define/contract (merge-advice-stacks trace-adv adv) (-> AdvStack? AdvStack? AdvStack?)
564   (let ([result (append (reverse (new-trace-advice (reverse trace-adv) (reverse adv))) adv)])
565     (begin ;(display "trace-adv: ") (display-list trace-adv) (newline)
566            ;(display "adv: ") (display-list adv) (newline)
567            ;(display "result: ") (display-list result) (newline)
```

```
568             result)))
569
570 ;; Without error checking
571
572 (define/contract (rw-resume-value-no-error v) (-> Value? Value?)
573   (deep-tag (all-tags v) (resumeV "dummy" 0)))
574
575 (define/contract (rw-replay-call-no-error f args adv sto tin) (-> Value? (listof Value?) AdvStack?
        Store? TraceIn? Result?)
576   (apply-with-weaving (rw-resume-value-no-error f) (map lift-trace-value args) (without-trace-advice
        adv) adv sto tin))
577
578 (define/contract (rw-call-no-error args adv sto tin) (-> (listof Value?) AdvStack? Store? TraceIn?
        Result?)
579   (rw-result-no-error adv sto (next-trace-state tin)))
580
581 (define/contract (rw-result-no-error adv sto tin) (-> AdvStack? Store? TraceIn? Result?)
582   (type-case State (trace-state tin)
583       [state (c adv-t sto-t tin-t)
584              (type-case Control c
585                [interp-init ()
586                             (error 'rw-result-no-error "Unexpected state")]
587                [app-call (f args)
588                          (type-case Result (rw-replay-call-no-error f args (merge-advice-stacks adv-t
                              adv) sto tin)
589                            (v*s*t*t (v-r s-r tin-r tout-r)
590                                     (rw-result-no-error adv s-r (next-trace-state tin-r))))]
591                [app-result (r)
592                            (v*s*t*t (lift-trace-value r) sto tin mt-traceout)])]))
593
594 ;; With error checking
```

```
595
596 (define/contract (rw-replay-call f args adv sto tin) (-> Value? (listof Value?) AdvStack? Store?
        TraceIn? Result?)
597   (type-case Result (rw-check-result
598                      (apply-with-weaving (rw-resume-value f tin)
599                                          (map lift-trace-value args)
600                                          (without-trace-advice adv) adv sto tin)
601                      tin)
602     [v*s*t*t (v-r s-r tin-r tout-r)
603              (v*s*t*t v-r s-r (next-trace-state tin-r) tout-r)]))
604
605 (define/contract (rw-check-result result tin-before) (-> Result? TraceIn? Result?)
606   (type-case Result result
607     [v*s*t*t (v-r s-r tin-r tout-r)
608              (if (< (length (tracein-states tin-r)) (length (tracein-states tin-before)))
609                  (let ([r (app-result-r (state-c (trace-state tin-r)))])
610                    (if (equal-values v-r r)
611                        result
612                        (error 'retroactive-side-effect
613                               (format "incorrect retroactive result: expected\n ~a but got\n ~a" r
                                       v-r))))
614                  (error 'retroactive-side-effect "retroactive advice did not proceed"))]))
615
616 (define/contract (rw-resume-value v t) (-> Value? TraceIn? Value?)
617   (let ([r (resumeV (value->string v) (length (tracein-states t)))])
618     (deep-tag (all-tags v) r)))
619
620 (define/contract (rw-call pos passed adv sto tin)
621                  (-> number? (listof Value?) AdvStack? Store? TraceIn? Result?)
622   (cond [(= pos (length (tracein-states tin)))
623          (type-case Control (state-c (trace-state tin))
```

```
624             [interp-init ()
625                          (rw-result adv sto (next-trace-state tin))]
626             [app-call (abs args)
627                       (if (andmap equal-values passed (map lift-trace-value args))
628                           (rw-result adv sto (next-trace-state tin))
629                           (error 'retroactive-side-effect
630                                  (format "incorrect argument passed retroactively: expected\n ~a but
                                       got\n ~a" args passed)))]
631             [else (error 'rw-call "Unexpected state")])]
632           [else (error 'retroactive-side-effect "retroactive advice proceeded out of order")]))
633
634 (define/contract (rw-result adv sto tin) (-> AdvStack? Store? TraceIn? Result?)
635   (let* ([t-state (trace-state tin)]
636          [_ (if (unbox verbose-interp)
637                 (begin
638                   (display "Weaving state: ") (display-state t-state (current-output-port)) (newline))
639                 '())])
640     (type-case State t-state
641       [state (c adv-t sto-t tin-t)
642              (type-case Control c
643                [interp-init () (error 'rw-result "Unexpected state")]
644                [app-call (f args)
645                          (type-case Result (rw-replay-call f args (merge-advice-stacks adv-t adv) sto
                               tin)
646                            (v*s*t*t (v-r s-r tin-r tout-r)
647                                     (prepend-trace tout-r (rw-result adv s-r tin-r))))]
648                [app-result (r)
649                            (v*s*t*t (lift-trace-value r) sto tin mt-traceout)])])))
```