AOD-Thorn: An extension of Thorn with the aspect-oriented programming paradigm

Wais Yarzi

Supervisor: Prof. Susan Eisenbach
Second marker: Prof. Sophia Drossopoulou

26th June 2011

Abstract

Object Oriented Programming hoped to solve many of the issues faced in procedural programming languages, by allowing a more natural description of the world in code through the use of objects. Code written in OOP is generally seen as often having superior modularity, reusability and readability compared to an equivalent procedural program.

Just as OOP concepts hoped to add a dimension to programming that would naturally solve some of the problems of procedural oriented programming techniques, so AOP hopes to naturally solve some of the issues faced by OOP, namely *tangling* and *scattering* that arise in a wide range of non-trivial OO programs.

Thorn is a new OO programming language targeted at the JVM, which currently has no AOP implementation. This work extends the language by modifying the Thorn interpreter to support AOP, utilising load time weaving with instantiable aspects. Further, we explore the idea of *transparently distributed* aspects, whose goal it is to make it easier to write aspect-oriented programs in distributed applications, by using a mechanism akin to a distributed heap.

Acknowledgements

I would like to thank the following people:

Prof. Susan Eisenbach for her guidance throughout the project. Her advice about the experiences of previous students doing similar projects has been immensely helpful in preventing me falling into several traps, and her Advanced Software Engineering course really helped while working on the implementation.

Prof. Sophia Drossopoulou for being my second marker, as well as giving me her Advanced Issues in Object Oriented Programming course, which was also immensely helpful when I was trying to work out the steps needed to implement my ideas for the language.

My family has been very supportive throughout my time at Imperial. Without their support, I probably would not finish this degree.

My girlfriend Elena, who has been a ray of sunshine in the last year, for giving me motivation when I was lacking it, and generally bringing a bit of sunshine when there were clouds over my head.

My friends who have constantly reminded me (with negative consequences on my academic work) that there is a world outside of university.

Contents

Chapter 1

Introduction

Thorn is a young language produced as a joint venture between IBM and Purdue University. It joins a growing trend of languages which target the Java Virtual Machine as the runtime environment.

Thorn is an object-oriented, imperative language, which uses the Erlang mechanism of no shared state for concurrency, utilising message passing of immutable objects between isolated *components* (processes running on separate JVM instances) for concurrent and distributed computations.

Furthermore, it uses a gradual typing system, which allows programmers to prototype dynamically typed, script-like programs that can be gradually evolved into statically typed "industrial grade" applications.

The gradual typing mechanism allows programmers to write applications in which it is possible to quickly adapt parts (i.e. parts without static annotations) which are often subject to frequent change, while bringing the maintainability and efficiency associated with static languages to design-stable parts of the application.

Thorn is very much a language which tries to understand the pressures faced by developers in real world projects, with tight deadlines and rapid evolution of requirements.

As most OO languages, Thorn suffers from the problems of *scattering* and *tangling,* as the OO paradigm cannot properly encapsulate certain *concerns* of an application using simply methods, classes and packages. This prevalence of scattering and tangling in OO programs can have a severely detrimental effect on modularity, which can often create a maintenance headache.

Aspect-oriented programming is widely regarded as a viable solution to decreasing the encapsulation limitations of the OO paradigm. If we think of procedural programs as one dimensional, we can think of OO as adding the second dimension, while AOP adds a third dimension. AOP is not a replacement to OO, it is meant as a supplement or partner to the OO paradigm, just as the OO paradigm was a supplement to procedural programming.

Within this project, there were 3 key requirements, ordered from the least to most esoteric.

The first requirement was to implement AOP constructs which are broadly common and available in existing popular AOP systems, such as AspectJ, Spring AOP or PostSharp.

The second requirement was to explore the idea that unlike in most AOP suites, aspects are no longer a layer above classes in a program. While it appears that early during the development of AOP it seemed as if aspects should be a layer above classes in the sense that classes cannot reference or instantiate aspects (i.e. aspects can modify how objects run, but not vice versa), the inclusion of certain features such as object being able to reference aspect instances in suites such as AspectJ, can be seen as strong evidence that this original layered design philosophy broke down as AOP became more well understood.

We make aspects not only referenceable, but also instantiable from objects. Furthermore, we explore the idea of selective and unselective aspects, which prove to give higher levels of control to a programmer who wishes aspect instances to apply to certain class instances. This effectively allows aspects to be turned on and off at runtime for a set of objects without resorting to clunky syntax. We have only found a single previous treatment of these ideas, in a research language called Ptolemy, which evolved out of a paper which discussed a mechanism known as Classpects[39], which unifies aspects with classes.

Finally, we explored the idea of aspects that apply to distributed programs. AOD-Thorn aspect instances are capable of communicating across separate JVM instances (including on separate machines), so that a programmer does not have to use any networking mechanisms in his aspects when the programmer wishes to advise a distributed system. For example, aspects are widely used for tracing a program execution and using our mechanism the same aspect can be used to trace the execution of a distributed application without the need for any modifications. We have not found any previous work emulating this idea.

The novelty of the project comes from:

1. Adding an AOP mechanism to Thorn
2. Transparently distributed aspects, which utilise an RMI like mechanism to advise objects in separate processes

Having said this, the requirement of user isntantiable, selective aspects is so rarely found in AOP suites that it may be classed as “novel”, even though we were beaten to the idea.

Almost all the ideas explored in this project can be utilised in AOP suites for other OO languages. If the language already has an AOP framework, it is likely that it does not allow the programmer to instantiate aspects and more likely that it does not allow programmers to use aspects in a distributed setting as easily as has been seen in this project.

1.1 Report structure

Chapter 2 should give a basic grasp of features of aspect-oriented programming (abbreviated as AOP) relevant to the project, as well as some approaches that are undertaken already. It also gives a description of the problem that AOP solves. Where appropriate, the chapter cites materials which the reader may find useful in order to learn more about AOP and the approaches to AOP.

Chapter 3 introduces some of the new features introduced in AOD-Thorn into Thorn, as well as some example scenarios of where these may be useful. This chapter should not be seen as a tutorial on AOP, so use cases did not receive a lot of attention (the reader is forwarded in chapter 2 to excellent literature on AOP use cases). As such, the examples provided are often minimal, simply showing off the language contructs introduced in AOD-Thorn.

Chapter 4 goes into some of the details of the implementation. However, it will only convey the main ideas, and will not go through all the complexity, as this is unlikely to be useful to the reader who will probably not want to go into line by line descriptions of how the implementation works. Some operational semantics of AOD-Thorn can be found here.

Chapter 5 evaluates the work done as part of the project. It attempts to describe both the strengths and weaknesses of the project.

Note The interpreter is a relatively large and complicated codebase, and probably the most difficult task during the implementation of the project was to understand the main architecture of the interpreter and then the understanding of how the most important tasks are done(such as how method invocation or field read and write is implemented).

The main difficulty came from the fact that the interpreter codebase has almost non existent documentation, and there isn't an active community around Thorn to ask questions of (in fact, at the time of writing, IBM who were developing the interpreter decided to stop developing Thorn).

As such, the report often links to specific parts of code (such as the package and class name) implementing functionality, so that this may serve as some kind of documentation to people who may work on the interpreter in the future (although it does not replace purpose built documentation, it should go a long way, and the author certainly wishes that previous projects had done this).

These links are assuming the commit numbered 3840 on the SVN repository where the source resides, dating from December 2010. At the time of writing, there are a further 10 commits, which were extremely minor, and therefore the links still hold (in fact, most the commits dealt with writing code in Thorn, rather than modifications to the interpreter).

Chapter 2

Background

2.1 Aspects

Separation of concerns is a phrase that is attributed to E.W.Dijkstra, after he mentioned it in a paper called "On the role of scientific thought"[13]. In the paper Dijkstra says: "It is what I sometimes have called “the separation of concerns”, which, even if not perfectly possible, is yet the only available technique for effective ordering of ones thoughts, that I know of. This is what I mean by “focusing ones attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant.”.

The use of the word aspect within this quotation is chiefly what gives rise to the phrase "aspect-oriented" programming, i.e. focusing only on a certain aspect of a program. Separation of concerns is an important principle widely in use in software engineering. Unfortunately, it is relatively rare to find a non trivial application written in an OOP language, which does not violate this principle repeatedly, that is, unless it uses some form of aspect-oriented programming.

Aspect-oriented programming[24] has gained some traction in industry, as a technique to decrease the issue of SoC violation in OOP programs. The idea is that code that makes up an application can be classed as solving the core concern or the cross-cutting concern of the application.

Core concern Code solving the core concern is said to be solving the problem that the application is being written for, directly dealing with the problem domain. For example, in a financial analysis application, the code which retrieves data, applies financial formulae and then outputs to some medium (such as the screen or the printer) is all code that deals with the core concern of the application. Functionality such as input validation is not a core concern, because it does not help us solve the problem that the application is being written for (financial analysis), even though it is an essential part of the application.

Cross-cutting concern A very significant portion of real world applications contain code which solves problems other than what can be classed as core concerns of the application. This can be, for example, security, logging, transaction handling, input validation and so on. When these pieces of code

Tangling A system with unnatural dependencies between its various modules is referred to as a tangled system. For example, business logic code which refers to the logging system has a tangle, as there is no natural link between business logic and logging.

Scattering Scattering refers to e.g. code duplication, where some piece of code is placed into many different places in the codebase. Note here that this code may be duplicated in terms of broad functionality, rather than duplicated completely. For example, logging code may be scattered throughout a codebase, each time logging different parts of it, but nevertheless encompassing only the concept of logging. In OOP programs such duplication can cause severe problems throughout the lifetime of the application.

One of the strongest characteristics of code that deals with cross-cutting concerns is that is often *scattered* throughout code that deals with core concerns, often in ways that violate the "DRY" (don't repeat yourself) principle, simply because it is sometimes difficult to encapsulate certain functionality within normal OOP languages. This is also known as the "**tyranny of the dominant decomposition**: the program can be modularised in only one way at a time, and the many kinds of concerns that do not align with that modularisation end up **scattered** across many modules and tangled with one another"[44]

While most introductory articles on aspect-oriented programming uses logging as the example for how aspect-oriented programming can help alleviate SoC infringements, this is in reality only a "hello world" application of what can be achieved with aspects. It has been noted that to truly demonstrate how aspects can help with real world applications, relatively long pieces of code need to be used as examples. This implies that while the advantages of the jump from OOP to AOP are not quite as quickly noticeable as the jump from procedural to OOP, it is nevertheless a significant one that can especially help with large code bases. Several books have been written on utilising Aspect oriented languages in real projects[21, 26, 42].

2.1.1 Examples

2.1.1.1 Common crosscutting concerns

There exists a significant body of research that catalogues common crosscutting concerns[31, 16]. There also exists significant research on aspect mining[30, 52, 7] (identification of code that would benefit from being aspectized) techniques as well as automatic refactoring techniques[52, 51, 63] to aid programmers in improving their architectures using aspects.

Authors experience with AOP The author had very limited experience with AOP prior to the project. However, he had seen AOP used in the following ways:

1. To reduce scattering / tangling in a large package of Java classes that representing the domain model (e.g. classes called Customer, Employee, Approver, etc..). There was a requirement to audit changes to objects of this package and previous developers had taken the route of simply adding logging code as the first line of every setters. This shows both scattering and tangling.

    (a) Scattering because there is a lot of repetition of calls to the logging subsystem (most of the logging statements were the same). Taking out the logging code and placing it into an aspect reduced scattering, as there was no need to repeat what was pretty much the same logging call every time. The code became more readable, as the programmer could concentrate on the business ideas, rather than non-core concerns such as logging.

    (b) Tangling because all these domain classes had a dependency on the logging subsystem. Moving the logging code out of the domain classes into the aspects increased modularity. The domain classes were used by other teams, who had different logging subsystems, or did not care about logging.

2. To enforce a rule across the application. The application had several hundred SQL queries, and used JDBC (Java DataBase Connectivity) to connect to a database and run the queries there.

    (a) The application had a per site policy, in that, each site (i.e. a bank branch) where the application was deployed should only see data for that particular site. They used a WHERE Site = site clause in the SQL for this. Sometimes people forgot to add the WHERE clause or somebody would incorrectly hard code the value of the site, and the application would break.

        i. AOP was used to intercept the calls to the JDBC library, and the SQL queries were modified to include the correct WHERE clause. This was a great scattering reduction as the where clause was no longer scattered throughout the queries.

2.1.2 Join points

A join point, in AOP terminology, is simply a "point in the control flow of a program"[59]. In other words, a join point is some event that happens during the execution of a program that we may wish to intercept and give *advice* to (advice on how to evaluate).

The various aspect-oriented platforms provide support for different types of join points, and which join points are provided depend on the underlying language as well as the languages used to implement aspecting. Nevertheless, the author identifies some joinpoints which are seen common to most AOP frameworks.

2.1.2.1 Common join points

Method call: A method call join point specifies that the joinpoint should be matched whenever a method is called, at the site of the callee rather than the caller. This means that the joinpoint context (context information about the jointpoint, e.g. which piece of code made the method call) will have a reference to code that made the call to the target method, rather than a reference to the target method itself.

Method execution: Similar to a method call, except that the joinpoint context has a reference to the target method, rather than the callee.

Constructor call: Similar to method call, but specifically for constructors.

Constructor execution: Similar to method execution, but specifically for constructors.

Object field read: Specifies that the joinpoint should be matched whenever a field matching some pattern has been read from. For example, we may write a diagnostic Aspect that counts the number of times a field is read during the execution of our application, and we can use the joinpoint context to find out which pieces of code do the reads and how often.

Object field write: Specifies that the joinpoint should be matched whenever a field matching some pattern has been assigned a new value. Similarly to the example for object field reads, we may write an aspect to find out which pieces of code are writing to a field, and how often (as well as the values that they are assigning).

Scope: It is common to see joinpoint languages have some sort of mechanism for defining scope for joinpoints. In Java, we may wish to only match joinpoints which are in a certain package, class or method. For example, we may wish to log the invocation of any setter methods for domain classes defined in a certain package.

2.1.3 Pointcuts

A pointcut is a "set of join points"[56] that describe when advice should be executed. In other words, a pointcut is a disjunction of predicates, which if true at a point T in the execution of the program, prompts the execution of advice at time T.

2.1.4 Advice

Advice is code that can run *before*, *after*, or *around* (instead of) the joinpoint. In other words, we are able to intercept a joinpoint, and then inject code before or after the code that the joinpoint references, or we can completely replace the code that the joinpoint references. The following are some examples of the usefulness of the 3 types of advice:

- **Before / after advice**: we may wish to enforce some *pre-* or *post-condition.* We can do this by injecting the appropriate code before or after the join point. We may also
- **Around advice**: we may wish to completely replace how certain functions work. Around advice can be used to aid in refactoring existing code, or it may be used in more sophisticated ways. For example, [53] demonstrates the rather elegant use of around advice coupled with Java annotations and reflection to make methods annotated with "@Asynchronous" to be executed asynchronously (i.e. the method is executed in a separate thread), which resulted in significant code simplification.
  - A second example may be to profile a method. The around advice would specify that a timer is to be started just before the method starts executing, and then the timer would be stopped just after the method finishes. In some programs (e.g. single thread), around advice can be emulated using a before / after pair of advices.

2.1.5 Aspect precedence

Non trivial applications utilising aspects may run into situations where multiple pointcuts match a given joinpoint. The correctness of a program may depend upon the order in which aspects are executed, making it an important concern for the programmer while writing the aspecting code. There has been work on solving this problem, with AspectJ following a certain set of conventions (e.g. if multiple pointcuts in the same aspect match a joinpoint, the first to be defined will advise the joinpoint first), as well as giving explicit priorities [35]. There has also been some work on parallelising advice (when it does not affect correctness)[64] as well as generally adding more flexibility to what is offered by i.e. AspectJ precedence rules[9].

In the project, an advice precedence model akin to AspectJ was adapted, i.e. it can be explicitly stated + conventions.

2.1.6 Weaving

An aspect weaver is a “tool used for integrating aspects with classes”[38]. In other words, the weaver is what combines code defined in aspect code with the core code. For example, when we write an aspect that provides e.g. before advice to a method, the aspect weaver will match the pointcut in the aspect with the method, and then somehow inject the advice before the code for the method.

There are two main types, compile/load time and run time (or dynamic) weavers.

How advice is weaved into the core code has significant performance implications on the applications, so it is important to get this right. The AspectJ team (which uses a bytecode level weaver, i.e. it reads JVM bytecode and modifies it) has stated: “We aim for the performance of our implementation of AspectJ to be on par with the same functionality hand-coded in Java. Anything significantly less should be considered a bug“[46]

2.1.6.1 Compile / load time weavers

Compile time weavers modify the bytecode of the application during the compilation stage (sometimes referred to as offline weavers). In Java, a compile time weaver such as the one found in AspectJ modifies the bytecode of *.class* files[19]. For this to happen, it needs to go through several stages:

1. Compile aspects to bytecode
2. Compile classes to bytecode
3. Match points in the control flow of the program with the pointcuts in the aspect definition files
4. Inject inlined advice bytecode, or a reference to advice bytecode, through i.e. a function call

Load time weavers do something similar, however they modify the classes while they are being loaded by the JVM. Obviously, load time weaving has a penalty on the time it takes for a class to be loaded, which has implications on the application startup time.

2.1.6.2 Run time weavers

Run time weavers, such as found in frameworks like JBossAOP or AspectWerkz, weave classes that have already been loaded by the JVM. In other words, a programmer can "dynamically plug and unplug an aspect in / from running software"[43]. As such, run time weaving brings with it functionality to the AOP world that static weavers do not have.

The biggest winner from the run time weaver movement is software that cannot go down (to be recompiled) but may require aspects to be loaded and unloaded.

[43]gives an interesting example. Lets say we have a critical web application that cannot go down, but that we wish to profile under heavy loads, when it starts to slow down. The static way to do this would be to write advice that checks the load of the application, and if it has reached some threshold, to start logging data. Unfortunately, this method adds a performance overhead onto parts of the application to be profiled at *all* times (instead of just when we log data), due to continuously having to check some predicate in order to decide whether to proceed or not. Clearly restarting to insert aspect code is not an option, as we may lose the state that caused the application to behave in a way that required profiling.

The run time solution to this problem is to simply not weave the profiling aspect until it is required. In other words, we *plug in* the profiling aspect when the load reaches a certain threshold. This should mean that the overhead when we are not profiling will be eliminated.

There has been significant work in utilising JIT compilation (by e.g. making modifications to the JVM) with run time weaving to bring the performance to as good as static weaving or even better performance, e.g. [17].

2.1.6.3 How dynamic weaving works

There are two main approaches towards dynamic weaving on the JVM, class reloading and proxies.

Class reloading On the JVM it is possible to, at runtime, modify the bytecode of a class and then reload that class, without having to restart the application[1]. Using this, we could use traditional bytecode weaving techniques (i.e. inject bytecode of advice before / after / around bytecode of joinpoint), and then ask the JVM to reconsider the definition of the woven class.

Proxies This is largely an approach taken by libraries (such as SpringAOP) as opposed to compilers like AspectJ. To weave advice into an object, the JVM reflection API is used to create an object with the same interface, which however, has before / after / around advice (in essence, Java classes are generated by the library and objects of the generated class are used as proxies for other objects). For example, if a class has method foo(){ X }, then a proxy with before advice would have method foo(){beforeAdvice(); X}. This type of weaving can cause significant overhead[25] (i.e. there is a level of indirection, a method call, which may not have been required before), however, this may be low relative to the code being advised (i.e. a method call is insignificant compared to database write, which may be advised).

A further, and more significant (and for some applications, crippling) limitation is that certain joinpoints cannot be implemented using proxies. For example, in Java it is not possible to write a proxy which intercepts field writes / accesses, or one that intercepts constructors (a proxy needs the object it is to be a proxy for to be constructed already).

2.2 Aspect oriented languages and frameworks

Before proceeding, it is important to note that much of the research and work on aspect oriented programming was done for Java, with the most mature AOP suites belonging to Java. The maturity of these meant that the work on this project is most heavily influenced by Java AOP frameworks. Having said this, some languages such as Fred[36] came with aspect-oriented features already built in, although these languages tend to be largely for research purposes and are generally not used in practice.

2.2.1 Two distinct types of AOP

There are 2 main distinct approaches to implementing aspect orientation on the JVM:

1. As a language extension
   (a) Translate code in extended language into underlying language (e.g. meta-AspectJ which translates to normal Java [20])
   (b) Modify the bytecode of the underlying language (e.g. AspectJ)
   (c) Optionally, make changes to the JVM to optimise it for AOP awareness (experimental versions of AspectJ[17])
2. Use features of the existing language (e.g. libraries like AspectWerkz, JBossAOP, SpringAOP)

2.2.1.1 Language extension

AOP implementations that are built as a language extension typically require a custom compiler.

Programs which utilise AOP technology implemented as a language extension need to be compiled (or translated) with the custom compiler (or translator) (e.g. ajc for AspectJ) rather than the standard one (e.g. javac for Java). As such, all existing programs written in the underlying language are typically valid programs in the derivative AOP language, but not vice versa. For example, all Java programs are valid AspectJ programs but not all AspectJ programs are valid Java programs (due to the language constructs of AspectJ).

While we can utilise runtime weaving with the language extension approach, it has traditionally been the case that such extensions relied on static weaving, as they did not want to lose the compile time type checks. Usually, language extensions can "enable better static checking"[20].

Using a language extension approach is that we can completely tailor the syntax to aspects. Using existing language features (such as annotations) may result in code which is more verbose than necessary (i.e. language extensions can lead to more concise code).

Finally, a language extension may be able to perform better optimisations than a runtime library[20].

Unfortunately, implementing aspects as a language extension also has drawbacks.

1. **Higher learning curve -** the programmer needs to learn new syntax.
2. **Modifications to the build process -** this appears to concern some developers, although sometimes for seemingly irrational reasons (e.g. the author was once told "its outside the convention we use in the company, no real other reason").
3. **Lock into language extension -** AspectJ is not a language extension which translates into Java, but instead its a bytecode modifier.
4. **Complicating the language implementation -** software projects are notoriously susceptible to collateral damage, i.e. breaking something that is seemingly unrelated to the feature being implemented. Adding new language features can break existing functionality. For example, in this project joinpoints were modified (i.e. the way methods run), which caused the interpreter to be more complicated. Further, if the language extension is not officially supported by the vendor of the runtime (i.e. if AOD-Thorn was not supported by the main developers of Thorn), then the extension must always be updated to the changes made by the vendor. People may be put off by this, as they may not trust that the extension

will be maintained as readily as the underlying language. For extensions not supported by the underlying runtime vendor, translators may be the better idea, as a programmer can remove a translator at any point and perform changes to translated code by hand.

2.2.1.2 Use existing language features (i.e. library approach)

In Java, there are implementations such as JBossAOP (or later versions of AspectJ) which utilise existing language features (namely annotations). One can simply annotate classes using an annotation identifying aspects, and then annotate methods as advice. Annotations or external files (e.g. XML) can be used to define pointcuts and so on.

The advantages of the library approach are the same as the disadvantages of the extension approach, and vice versa. However, we should take some notes:

1. **Lower learning curve -** programmers do not need to learn less new syntax, and they can use concepts (e.g. annotations) that they are already familiar with. This reduces the load on the programmer who is new to AOP, as opposed to language extension approaches. Having said this, things like joinpoints will require new syntax for the programmers to learn, and things like the XML schema for JBossAOP are almost like new languages anyway.

2. **No need for an external compiler -** one of the biggest complaints from programmers using language extensions is the need to modify their build, which can be a relatively complicated process for large existing applications. A further problem is that language extensions are usually mutually exclusive, with users not wanting to lock themselves down to a single extension for the lifetime of the projects. Not using a language extension means it becomes possible to aspectize code without using an extra compiler, however, a preprocessor may still be used, for example for static type checking.

2.2.2 AspectJ

AspectJ was publicly released in 2001 by researchers from PARC[23] (Palo Alto Research Center, a division of Xerox) and several universities, including Gregor Kiczales who was one of the authors of the original Aspect-Oriented Programming paper [24]. It is seen by practitioners as the most popular method for using aspects in Java (or the JVM as a whole).

AspectJ was at first purely a *language extension* of Java. The language extension has very good IDE support (as described in section 2.2.2.1). As a language extension, AspectJ has a complete syntax of its own, however, it is also able to function completely through annotations, without having to resort to the language extension. This flexibility, as well as the large body of research work that has gone into AspectJ (a very significant portion of research that goes into aspect oriented programming in general used AspectJ in some way, and made extensions to it), is probably what makes AspectJ currently the most popular AOP implementation.

While AspectJ was originally at PARC, it later on moved to the Eclipse foundation, and has been heavily developed by Spring Source[50].

2.2.2.1 Tooling support - AJDT

AJDT[10] (AspectJ Development Tools) provides a multitude of functionality designed to make it easier to write aspect-oriented programs on Eclipse.

Some of the functionality that AJDT provides:

1. **Outline view[?]** - show which pieces of code are affected by a pointcut.
2. **Annotations -** link from code back to aspects that advise it
3. **Aspect visualiser -** visually represent how the aspects are affecting the codebase (i.e. when editing a method, it would show which aspects are going to advise it at runtime)
4. **Debugger -** debugger that allows to step through advice code
5. **Aspect aware refactoring support -** e.g. type renaming within Eclipse is a trivial exercise. However, when joinpoints have e.g. wildcards, this becomes more problematic. AJDT tries to alleviate these problems, by letting the programmer know when pointcuts which used to match the refactored joinpoint no longer match them.

One of the biggest issues with the takeup of AOP in industry is that code can be somewhat harder to read for developers not used to AOP, because it becomes difficult to understand the program flow if, encapsulated units of computation such as methods, are no longer in a single place (but rather, methods would have their definition alongside of advice, which may lie inside complicated pointcuts).

Clearly tool support like provided by AJDT is very useful, and possibly vital for the uptake of a language that supports AOP.

2.2.2.2 AspectJ example

The author provides a “hello world” type example written in AspectJ, which should give the reader a taste of the syntax employed by AspectJ. Comments about how the aspecting works are inlined with the aspect code.

Listing 2.1: Employee.java

```java
package com.company.employeebeans;

public class Employee {

  public long employeeId;
  private String name;
  private String address;

  public Employee(long employeeId, String name,
      String address){
    this.employeeId = employeeId;
    this.name = name;
    this.address = address;
  }

  public void setAddress(String address){
      this.address = address;
  }

  public String getAddress(){
      return address;
  }
}
```

Listing 2.2: Address.java

```java
package com.company.employeebeans;

public class Address {

  private String firstLine;
  private String secondLine;
  private String city;
  private String postcode;

  public Address(String firstLine,
      String secondLine, String city,
    String postcode) {
    this.firstLine = firstLine;
    this.secondLine = secondLine;
    this.city = city;
    this.postcode = postcode;
  }
}
```

Listing 2.3: LoggingAspect.aj

```
package com.company.employeebeans;

import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public aspect LoggingAspect {

  private Logger log = Logger.getLogger(
                        LoggingAspect.class);

  public LoggingAspect(){
    log.addAppender(
          new ConsoleAppender(new SimpleLayout()));
  }

  // match whenever we have a setter method in
    // any class
  // in the com.company.employeebeans package
  pointcut beanSetterCalled(): execution (* *.set*(*))
    && within(com.company.employeebeans.*);

  // after beansetter is called, log the arguments
    // the setter was called with
  after() : beanSetterCalled(){
    Class<?> beanClass = thisJoinPoint.
                              getThis().getClass();

    String signatureName = thisJoinPoint.
                                getSignature().getName();

    Object[] args = thisJoinPoint.getArgs();

    log.info(beanClass.toString() + " has executed "
            + signatureName + " with arguments "
            + stringifyArgs(args));
  }

  private String stringifyArgs(Object[] args){
    StringBuilder sb = new StringBuilder();
    for (Object arg: args) {
      sb.append(arg);
      sb.append(" ");
    }
```

```
        sb.deleteCharAt(sb.length()-1);
        return sb.toString();
    }

}
```

2.2.3 JBossAOP

JBossAOP was one of the first aspect-oriented frameworks for the Java platform to purely work as a library rather than a language extension. While it is a mainly marketed as a library, it also supports compile / load time weaving (using the aopc compiler). Aspect code (including advice) is written in plain .java files, either through extending an interface in the framework or not, which influences the amount of configuration that has to be done later. Pointcuts are defined using XML, or they can be defined using annotations.

JBossAOP has two main workflows, one that uses XML files and the second that uses annotations[49].

2.2.3.1 Interceptor

The easiest way to define advice in JBossAOP is to extend the Interceptor interface.

```
import org.jboss.aop.advice.Interceptor;
public interface MyInterceptor extends Interceptor{}
```

The interceptor interface is as follows:

```
package org.jboss.aop.advice;
import org.jboss.aop.joinpoint.Invocation;

public interface Interceptor
{
    public String getName();
    public Object invoke(Invocation invocation)
                                  throws Throwable;
}
```

The invocation will have contextual information on the joinpoint, similar to the "thisJoinPoint" keyword in AspectJ, for example, the name of the method called or the arguments being passed into the method (when dealing with a execution / method call pointcut).

For the following examples, we use the same Employee.java and Address.java as in the AspectJ example, however, we replicate the functionality of LoggingAspect.aj using JBossAOP techniques.

The first way to replicate the LoggingAspect.aj functionality is through extending the Interceptor class as follows:

Listing 2.4: LoggingInterceptor.java

```java
public class LoggingInterceptor implements Interceptor {

  private Logger log = Logger.getLogger(
                         LoggingInterceptor.class);

  public LoggingInterceptor(){
    log.addAppender(new ConsoleAppender
                   (new SimpleLayout()));
  }

  @Override
  public String getName() {
    return "LoggingInterceptor";
  }

  @Override
  public Object invoke(Invocation invocation)
                         throws Throwable {

    MethodInvocation mi = (MethodInvocation) invocation;
    Class<?> beanClass = mi.getTargetObject().getClass();
    String methodName = mi.getActualMethod().getName();
    Object[] args = mi.getArguments();

    //stringify args as before
    log.info((beanClass.toString() + " has executed "
        + methodName + " with arguments "
        + stringifyArgs(args));

    return null;
  }
}
```

We can see from this example that the advice is given in pure Java and pointcut definitions are externalised, with no need to extend the definition of a class in order to support AOP.

The XML workflow The first approach, common to many frameworks in Java, is to externalise configuration information into an XML file. We can think of the pointcuts as the configuration and the execution of advice as the thing needing the parameters in the configuration.

```
<aop>

   // Define the pointcut for
   // setter method calls in all classes in
   // the employeebeans
   // package
   <pointcut name="EmployeePkgFieldSet"
     // the expression is a string
     // the XML schema cannot verify
     // whether its correct syntax.
     expr="execution(public * com.company.
                    employeebeans.*->
                    set*(..))"/>

    // when the pointcut EmployeePkgFieldSet
    <bind pointcut="EmployeePkgFieldSet">
       <interceptor
         class="com.company.employeebeans.
                LoggingInterceptor"/>
    </bind>
</aop>
```

Annotations Pointcut configuration can also be done using annotations, inside Java classes. For example, the interceptor is annotated with with the @Bind annotation.

```
@Bind(pointcut="execution" +
    "(public * com.company.employeebeans.*->set*(..))")
public class LoggingInterceptor implements Interceptor {
```

XML vs Annotations The real (more general) argument here is whether to add aspecting information (such as @Bind) along with the core code, or whether to externalise this information out of core code, i.e. into an external file.

The argument may really be whether we are willing to accept the pollution of code by aspecting constructs. For example, if we have an interceptor that we wish to reuse in another program, it is most likely impractical to use as is, the annotations will have to be changed (if a different team uses our code, they need to branch it, giving them a maintenance task).

However, it is relatively tedious to keep a completely separate file, and may reduce programmer productivity (or they may simply not use the AOP if they feel it slows them down too much).

2.2.4 Alternative approaches to separation of concerns

There are a number of alternative approaches to separation of concerns, which do not necessarily involve an explicit AOP methodology / framework.

Many dynamic languages have features which make it easier to write code that can separate the crosscutting concerns out of the core concern code. Further, languages with first class functions can also make this separation easier.

For example, in Python, a relatively small amount of syntax is required to emulate before, around or after advice for functions, since it is a dynamic language with first order functions, simply wrap a function in another function. Something similar can be done in Java, using reflection, but it is much more syntactically expensive. Of course this approach still suffers from the problems of proxies in Java, an extra level of indirection as well as not being able to intercept e.g. field accesses, although there is no longer the problem of not being able to intercept constructors, as we're not wrapping entire objects but methods only. However, there is still a need for writing code that will do the wrapping (i.e. checking objects for joinpoints etc), along with writing the advice etc.

The general point from the Python example we can take away is that while it is possible to emulate AOP using commonly, non AOP specific features of the language, it may not be as purpose made / easy to use as if we make a tailor made AOP implementation. "Advanced separation of concerns techniques are feasible and useful for dynamic, lightweight languages" [12]

In Python, several AOP frameworks (e.g. AOPY[33], aspects.py[22]), have appeared, which mostly utilise the wrapper approach, but making it easier to do so

An interpreted language, AspectLua (an extension of Lua), was also developed[11], without modifications required to the interpreter (the project also works on an interpreter). The interpreted nature of Lua (which is a dynamic, first class functions language) meant it was relatively easy to extend it for AOP. More interestingly, the interpreted nature of the language meant that it was very simple to implement dynamic weaving (i.e. redefine or add new aspects at runtime). However, it should be noted that by attempting to extend Lua without modifying, they were restricted to the features available in Lua, which had no viable mechanism for intercepting field writes. It is not always realistic to expect users to use the setters and getters for fields in the same class. Generally, there are AOP features which are difficult or impossible to implement using language extensions for most languages, which is why bytecode modification (e.g. AspectJ) / runtime system modification (e.g. this project, a modification of the interpreter) are sometimes required.

2.2.4.1 Metaclasses

A metaclass is "a class whose instances are classes"[60]. A metaclass α, in essence, allows us to define the behaviour of a class β, without us changing the code of β, and not requiring the use of the inheritance mechanism to achieve this. Defining the behaviour of β through the use of metaclasses is more flexible than doing so through inheritance, as αcan modify the behaviour of any class, rather than its superclasses (as is the case with inheritance), although what behaviour can be changed by $\alpha$is highly language dependent.

A metaclass can be thought of as a class factory, which takes as input a class and produces another class, whose instances we can create. They can therefore implement aspect oriented programs in a similar manner as proxies, in the sense that a class with the same methods / fields is produced, however, methods are modified with advice. However, they generally have no need to introduce the indirection associated with proxies, and do not suffer some of the limitations that proxies have, i.e. not being able to advise constructors, although they still suffer from limitations such as not being able to intercept field accesses (as has been the case with all the alternative approaches we have seen, as this is generally not a language feature), unless the language automatically gives us setter / getter methods for field accesses, i.e. when we write a setter / getter, it is merely overriding the field access, as opposed to adding a new way to access those fields where the fields can still be accessed directly.

2.3 Thorn

Thorn was developed mainly at IBM and Purdue university. The Thorn language specification isn't finalised and was under heavy discussion before the language was abandoned. Thorn targets the JVM platform, the interpreter is written in Java.

The interpreter is named Fisher, which was actively developed up until winter of 2010 (after which it was abandoned at IBM) and a bytecode compiler which was developed until 2009 (according to SVN logs where the source for Thorn lives).

Given that the interpreter had features described in the newest language specs (and was still being developed when the project started, therefore was more likely to get support from the developers), it was chosen instead of the compiler. Unfortunately, the main developer working on the Thorn interpreter made his last SVN commit near the end of December 2010, with some extremely minor changes being made by his replacement for the next 3 months, after which no further work was done.

2.3.1 Thorn interpreter structure

2.3.1.1 Parser

Strictly speaking, a parser is a module which takes as input a string, and produces as output a set of tokens, which are defined in the grammar of the language. One can then check that the tokens follow a grammar specification, place them into an AST and finally evaluate the AST to get results from the code contained within the input string.

The fisher parser is defined in a JavaCC[1] grammar and the AST nodes are defined by a Python script. While the AST classes could have been generated by Javacc, the fisher developers chose to build their own Python script for it.

JavaCC grammar definition The Thorn grammar is specified in the fisher.parser package in a file called grammar-fisher.jj. This file is passed to JavaCC, which generates most of the classes in that package, including FisherParser (the main parser), Token and so on. The fisher.parser package defines the parser, however, AST nodes are defined in the fisher.syn package. The majority of the fisher.syn package is generated by a Python script defined in /src/syntax/ast-fisher.py.

When the parser runs on a given .th file (the file extension for Thorn source files) or on an input string given in the REPL environment, it produces an AST whose nodes are defined in the fisher.syn package. Once the AST is generated from the input, it is visited by visitors in the fisher.syn.visitor package. The visitors preprocess and evaluate the program.

The majority of the rules in the grammar only require a lookahead of 1, however, some rules have this lookahead (locally, for that rule only) set to higher lookahead values, up to 5 to avoid ambiguities in the grammar.

The JavaCC input file is relatively easy to understand for someone coming from a Java background. Lets look at an example section of the grammar definition file:

```
If /*&*/If():
  { Token start, end;
    Cmd test, thenarm, elsearm=null;
    boolean reallyUnless = false;
  }
{
  (   start = <IF> {reallyUnless = false;}
    |start = <UNLESS> {reallyUnless = true;}
  )

  <LPAREN>
  test = BiggishExp()
  <RPAREN>
  thenarm = Stmt(start.image,null) {end = thenarm.end;}
  [
    // We've got a dangling-else problem here.
    // JavaCC's default behavior is to do the right thing.
```

[1]http://javacc.java.net/ - described as a “Java compiler compiler”

```
    // But it prints a warning because of the ambiguity.
    // cf. https://javacc.dev.java.net/doc/lookahead.html
    LOOKAHEAD(1)
    <ELSE>
    elsearm = Stmt(start.image,"else"){end = elsearm.end;}
  ]
  {
    return new If(start, end, test, thenarm,
                  elsearm, reallyUnless, false);
  }
}
```

It should be fairly obvious about how the above listing works. Provided below are some notes:

If /*&*/If(): Define how the If statement is parsed. If() is called from other definitions when they wish to consume an If statement. Upon consuming an if statement, the parser will return an AST instance of type fisher.syn.If , which could be used in another AST.

{Token start, end;...} The first pair of braces are used for variable declarations that will be used in the rest of the definition. In this example, the Java code in the second pair of braces (Java code is enclosed by curly braces in the second pair of braces of the definition) may refer to Token objects called start and end.

<LPAREN> Try to consume a LPAREN token defined earlier as "(". The token can be specified using a regular expression, rather than just as a literal.

[...] Indicates optional tokens, so in the above listing, the else part is optional.

return new If(start,end,test,thenarm,elsearm,reallyUnless,false); This returns an instance of the *If* AST

AST tree / Python script The /src/syntax/ast-fisher.py Python script generates the classes in the fisher.syn and fisher.syn.visitor packages. The code generating script includes a class CLS, instances of which get turned into the Java classes found in fisher.syn package.

When the parser has produced an AST, it is visited by the fisher.syn.visitor visitors. Following this, the AST is evaluated by fisher.eval.Evaller (one per each thread).

The Evaller has different strategies on how to handle an AST node depending on the node type. If the node type is a subclass of fisher.syn.Cmd (most major pieces of syntax are) then the nodes are visited by an instance of fisher.eval.Computer.

The Computer object is asked to visit the nodes and is supplied with a Frame-like object which contains environmental information required to evaluate an AST, such as variables (and their values) in the current scope.

2.3.1.2 Seals / sealant

The sealant is one of the visitors in the interpreter. The purpose of the sealant is to give seals to nodes of the AST. Every identifier in Thorn has a Seal associated with it (and therefore, every method, variable, class and so on).

A seal has a few important uses in the interpreter:

1. **It serves as a unique ID for each AST node.** For example, an object in Thorn is represented in the ObjectTh class. An ObjectTh has fields, to which we can assign values. To do this, we need to unambiguously identify the field of the ObjectTh. The seal of the field can give us this unambiguity, as the sealant will produce a seal for each field when it visits the class declaration in the AST (and subsequently, the field AST nodes).

2. **Gives some scoping information -** each seal has a reference to the seal of the static thing which encloses the identifier. For example, in **module Mod { var x; },** the seal of the container of variable x is the seal of module Mod (and the seal for Mod also has a back reference to all seals that it is the container for). This helps in scope checking. For example, (a highly simplified example) if we have code of the form **module Mod { var x; println(x);},** then we can quickly check whether x in the println statement is a variable in the scope of Mod by asking Mod if it is the container for that variable.

Seals have been described (in a previous project on Thorn) as similar to the De Bruijin index[37], although this is not quite correct, as the De Bruijin indexes[8] are mainly used to check for α-equivalence in λ-calculus, while a Seal is a far more general purpose entity.

2.3.1.3 Frames

A fisher.eval.Frame is essentially a stack frame. The root frame contains a reference to *PredefinedIdentifiers,* which is an enum class that has seals for some built in functions (that don't need to be imported, e.g. println) or built in variables, such as *argv* (command line arguments passed into the Thorn script upon startup). For example, during method calls, frames are used to store an implicit *this* parameter, as well as the values of arguments to a method call. Its interesting to note that this approach is one of the thing that makes the interpreter much slower than the compiler, as the creation of frame objects is far slower than creating a stack frame on the JVM.

2.3.1.4 Evaller / Computer

The Evaller and Computer are two important visitors.

An Evaller basically evaluates the AST tree to give results. For example, the expression 1 + 1 would have 3 nodes on the AST tree, and the Evaller will modify the tree to replace the 3 nodes with one node, i.e. the value 2. In the interpreter, there is one Evaller object for each thread. This is useful, as the different threads do not have shared state (after all, the whole language has no shared state). This gives the JVM some opportunity to take some advantage of parallelism, as the problem of evaluating separate AST trees is *embarrassingly parallel.*

The Computer is what traverses the main AST, and then passes subtrees of the AST to the Evaller to be evaluated.

2.3.1.5 Pre-defined functions

As mentioned in the section on Frames, Thorn has a number of predefined functions, which do not need to be imported in order to be used.

For example, it is perfectly legal to have a .th file as follows:

Listing 2.5: Legal.th

```
module M{
  println("Hello world!");
}
```

Notice here that, unlike languages such as Java, we do not import the println function. This is because the constructor of the root frame will traverse the *PredefinedIdentifiers* enum, which has the seals of all predefined functions (including println). This means that the evaluator (which uses frames during evaluation) will always be able to refer to the the println seal and thus execute it.

It is relatively easy to add new pre-defined functions.

For example, the entry for the *println* function in *PredefinedIdentifiers* is as follows:

Listing 2.6: PredefinedIdentifiers.java

```
...
// make the seal for println, the seal
// should point to the PrintlnBIF class
predef("println", SealMaker.ofPredefFun("println"),
        PrintlnBIF.class),
...
```

If we look at *PrintlnBIF.java*, it should become obvious how all of this comes together:

Listing 2.7: PrintlnBIF.java

```
public  class  PrintlnBIF extends BuiltInFunctionTh  {

    // apply is in BuiltInFunctionTh
    // a Thing is the Thorn equivalent of java.lang.Object
    @Override
  public Thing apply(Thing[] args, Framelike ignoredFrame,
        Evaller evaller, Syntax src) throws FisherException {
        // join the arguments passed into println
        // and print out the resulting String to the console
    System.out.println(Bard.sep(args, ""));
    System.out.flush();
    return null;
  }
}
```

This shows that we can very quickly add new predefined functions simply by appending to the *PredefinedIdentifiers* and then extending the BuiltInFunctionTh. This approach was used throughout the project for prototyping new functionality (for example, adding objects to aspect instances). There do not seem to be any rules in Thorn for what is a good candidate for a predefined function and what should be built into objects (i.e. using e.predefFun(...) syntax).

2.3.1.6 Thing / ObjectTh

An ObjectTh is basically a Thorn object. It has a reference to a *ClassDynamic* (which contains information about the class of the object, i.e. method AST trees), as well as a map *Map<String, Frameable> fields*, mapping from field names to field name values (a Frameable is an interface for things which can be placed in a Frame). It is worth noting here that while the map uses String keys explicitly, implicitly it uses Seals. This is because the toString() version of a Seal is used as the key into the fields map.

A Thing is what all runtime Thorn objects derive from. Note that in Thorn, we do not only have ObjectTh instances as runtime objects, however, we also have such objects as Javaly function objects. These do not derive from ObjectTh (as they are not Thorn objects), however they are runtime objects.

2.3.1.7 Javaly

Javaly is the mechanism which Thorn uses to bridge between it and Java, so that it may make calls to libraries written in Java. This type of functionality is useful for languages targeting the JVM, as they do not need to implement an entire API themselves. Instead, they can write API wrappers which can use the extensive and mature API of Java. We can get an idea of how the Javaly mechanism works just by looking at the constructor of the *JavalyFunImpl* class:

Listing 2.8: JavalyFunImpl.java

```
...
import java.lang.reflect.Method;
...
    // Java method, using reflection
  public final Method method;

    // an AST tree representing a
    // javaly function
  public final JavalyFun decl;

  public final int arity;

  public JavalyFunImpl(Method method, JavalyFun decl)
     throws FisherException{
    super();
        // check whether the running script allows
        // javaly functions, give an error if not
    Security.sandbag("No javaly funs allowed.");
    this.method = method;
    this.decl = decl;
    this.arity = decl.formals.size();
  }
...
```

Javaly works through the use of reflection (in the method call case, through the use of *java.lang.reflect.Method*). The interpreter also has a sandbox mode (switched on through a command line argument), which prevents it from calling Javaly code.

2.3.1.8 Components

Thorn follows the philosophy that shared state should not be allowed in concurrent programming, and follows the Actors model, i.e. it achieves concurrency through message passing.

A Thorn actor is called a *Component*, and it is implemented by the *fisher.runtime.ComponentTh* class. Components are ran on different JVM instances, and they communicate to each other through sockets (and serialising messages through the use of *java.io.ObjectOutputStream*).

Having Components run on different JVM instances was originally intended to improve system robustness by "isolating failures"[6] (following the ideas of Erlang), although the robustness afforded is somewhat limited without a replication mechanism. One of the design ideas with running components on separate VMs was that if a Component goes down, it may be restarted by a *supervisor* process[28] (as in Erlang), although this functionality isn't implemented in Thorn.

However, Thorn follows an "evolve as you go along" philosophy. The Components mechanism is a good fit for the philosophy, as it allows one to move from a concurrent system to a distributed system with minimal effort, as one can simply run the components on different machines. Since the components communicate through sockets even when running on the same machine, moving to a distributed architecture requires no programmer effort and adds no overhead other than the overhead latent to distributed systems, namely communication latency over the network.

Chapter 3

The language

This chapter introduces the extensions to Thorn that were made as part of AOD-Thorn. The section assumes a familiarity with Thorn (possibly gained by studying the Thorn OOPSLA 2009 paper[6]), although the examples should be straightforward to read with even minimal (or no) knowledge of Thorn.

3.1 Aspect-orientation (vanilla)

The first step of the project was to add aspect orientation to Thorn. The author decided to focus on simple method calls and field accesses (reads and writes), as these are possibly the most important features in any AOP language, and in fact, there aren't many more, except some rarely used features such as intercepting certain patterns of executions, i.e. by looking at the call stack.

There are many sources showing elegant solutions to problems in software engineering (some excellent sources for inspiration:[35, 18, 26, 27, 42, 21]). As such, the author assumes that the reader is familiar with the solutions that AOP can provide to a project (even if such familiarity is limited to a specific framework such as AspectJ, the project was not about researching AOP use cases after all), so no complicated and long examples will be provided (which is where AOP really shines), but merely somewhat trivial (easy to understand) examples that should showcase how to use the aspecting extensions.

3.1.1 Motivation

Just as in other OO languages without aspects, Thorn applications are vulnerable to tangling and scattering due to the difficulty in separation of concerns.

The project first started out as an investigation on adding fault-tolerance features into the language, through the use of checkpointing certain variables at given points. The original requirements were to add syntax into the language that would ensure that annotated variables were periodically checkpointed onto the local disk, however, the author argued that this could be an inferior approach to writing a library that utilised Aspects that intercepted field writes and then checkpointed them to the disk, as this would:

1. Not require the pollution of checkpointed classes with the non-core concern of checkpointing (a large scattering and tangling problem)
2. Not require the addition of further keywords to the language. Fault tolerance is a bit of a niche, and to add it as a language feature would require significant thought in order to be made as flexible as possible, for e.g. different checkpointing algorithms (the user may for example, decide that checkpointing should output to an xml file), which would be difficult if it was a language feature instead of having multiple competing libraries to do the job in a way tailored for a specific domain. However, languages such as Erlang do have built in features for fault tolerance. It was really the authors wish to build something more general, which would complicate the interpreter less, and allow the language community to build such features as fault tolerance more easily.

It is easy to see how one could write a checkpointing library that used field write interception, and advised that the fields are written to the local disk, from where the values for the fields could be read from when needing to restore a checkpoint.

Such an example should show why Aspect orientation should prove to be useful in Thorn, i.e. making it easier for the users of Thorn to implement certain types of frameworks (and tools), which could have a profound influence on how Thorn evolves (for example, Ruby on Rails was a framework which had a huge influence on how Ruby was perceived in the market).

Certain types of tooling also benefit from aspecting technology, i.e. diagnostic / monitoring, debuggers and profiling tools can be implemented much more easily[4] if the language has AOP functionality (diagnostic / monitoring: simply give advice to joinpoints to log parts of the programs in need to be monitored, profilers: inject timing code to joinpoints, debuggers: i.e. inject code which will report the values of fields when asked to by the debugger). Making it easier to write tools like this increases the chances of the tools being written, which in turn increase the chance of the language being marketable.

3.1.2 Vanilla aspecting

The vanilla aspecting portion of the project consists of intercepting method calls and field accesses and then injecting code (advice) at the given points, with some attention paid to the efficiency of implementation (although the Thorn interpreter is by no means high speed).

3.1.3 Scope

When defining joinpoints, it is often useful to be able to narrow down the joinpoint to a certain scope, i.e. a package or a class. For example, we may wish to log all setter methods in a given package which contains our business objects.[1]

This type of scoping is achieved through the use of regular expressions, in the same flavor as those used in the standard Java regex API[2], although it is slightly modified for the pointcut domain. Since every identifier (and therefore, every method / variable) has a seal (as described in section 2.3.1.2) attached to it, we can use regular expressions on the String representation of the seal to do scope checking. The seal has a toString() method, which simply returns a cached copy of the string representation of the location of the identifier (location here means the scope, i.e. the the package, class, identifier name) in the format of *packagename.classname.identifier* (for example com.company.businessobjects.Employee.setSalary for the method setSalary(..)).

This allows us to use wildcards to match parts of the seal string. For example, if we wish to write a joinpoint that matches all the setter methods in every class of the package com.mycompany.businessobjects then we can use a joinpoint of form:

```
(com.mycompany.businessobjects.*.set*)
// notice here that we use . instead of \.
// as would be expected if we were using pure
// Java regex patterns
```

When processing the regex pattern, in AOD Thorn all '.' characters are replaced by "\.". This is to avoid matching e.g. *comXmycompany* (in the above regular expression), without needing to escape the '.' characters. This was done because the '.' character is most often used in its literal form, rather than as a wildcard when defining pointcuts.

3.1.4 Method interception

To intercept a method call, we need only 4 entities:

1. Advice location - can be before, after or around. Advice location specifies where the advice code should be injected: before the joinpoint, after the joinpoint or instead (around) of the joinpoint (although around advice can still execute the original joinpoint).

[1]A business object is a "type of an intelligible entity being an actor inside the business layer in a n-layered architecture of object-oriented computer programs."[55]

[2]please see the javadoc for java.util.regex.Pattern at http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

2. A pointcut - with a set of joinpoints, which describe when advice should be executed.

3. The arguments passed into the advice, i.e. method arguments

4. The advice - a set of statements that define the code to be injected

Learning by example is an efficient method to learn a new language, so lets look at an illustrative example that shows how to use method interception in AOD-Thorn:

```
aspect SensorLog{

val alarmThreshold = 0;
val emergencyThreshold = 2;

var alarmsActivated := 0;

def log(msg) = println(msg);

before: exec jp[(com.sensors.tempsensors.IRSensor.sense)]
             jp[(com.sensors.tempsensors.ThermoCoupleSensor.sense)]
             args(arg){
      azpct.log(this.name + " sensed: " + arg);
      if(arg > azpct.alarmThreshold
           && arg <= azpct.emergencyThreshold){
         this.activateAlarm("Notify authorities");
         azpct.alarmsActivated++;
      } else if(arg >= azpct.emergencyThreshold){
         this.activateAlarm("Evacuate immediately!");
         azpct.alarmsActivated++;
      }

      //here for illustration only:
      println("sense called on: " + this.name +
               " by: " + origin.name);
}}
```

The above aspect corresponds to a fictional situation, where we have two sensor classes (infrared and thermocouple) that both have a method on them called sense (with a single argument). For some reason we are not allowed to modify the sensor classes nor override the sense method (lets forget about why for now[3]), so instead we use the advice defined above. Lets go through the interesting points:

[3]this illustrative example only shows the features of AOD-Thorn, not when to use aspects. Infact, actions such as activating an alarm would be classed as one of the core concerns of the class, and thus not classically what we would put into an aspect).

before: exec - this piece of syntax basically says "execute the advice before one of the methods in the set of joinpoints are executed"

jp[(regex1)] jp[(regex2)] - a joinpoint is defined by the keyword *jp* followed by a regular expression within square brackets. As noted before, a pointcut is a *set* of joinpoints, and therefore the language allows a list of joinpoints to be defined for a pointcut. It is worth noting here that as the full Java regular expression syntax is allowed, we could define a set of joinpoints using a single regular expression (i.e. through the use of the | operator), however the author felt that this would not be as readable as explicitly writing out the joinpoints (although the previous option is still available)

args(arg) - the args keyword, followed by a set of identifiers within round brackets, specify the arguments of the method being called. These arguments are available to the advice code.

azpct.log(...) - the *azpct* keyword is like the *this* keyword, however it refers to the aspect object rather than the object where the joinpoint is located. This particular expression calls the log method within the aspect object.

this.name - the *this* keyword refers to the object where the joinpoint is located. Infact, it is as if had written the this keyword in the original method that we are intercepting (the keyword is injected). [4]This particular expression refers to the name field of the *this* object.

origin.name - the origin keyword refers to the object which referenced the joinpoint. In other words, if object αhas a reference to objectβ, and αcontains code of the form $\beta.meth(,,,)$ or $\beta.field$, then we refer to βas the target (referenceable by the *this* keyword from within advice), and we refer to αas the origin. The expression origin.name evaluates to the name field of the origin object.

3.1.4.1 Some notes

exec The exec keyword tells us that we want to match on method calls, and is reffered to as a *pointcut primitive.* Other pointcut primitives include *set* / *get*, used for field access interception.

3.1.5 Field access

Intercepting a field access is very similar to intercepting a method call.

[4]It should become apparent from this that the azpct and this keyword imply a mechanism of cooperation between the aspect object and the object with the joinpoint. This is because advice inside the aspect object can always refer to the object containing the joinpoint method through the *this* keyword, while the object which has had advice injected into it has a reference back to the aspect object where the advice comes from (how this is achieved will be revealed in the implementation section).

Example:

```
  before: set jp[(*logged*)] args(val){
    println("Set on logged var in object: " + this.name
                + " by: "
            + origin.name +".\n
         Var now has value: " + val);
  }
```

It should be fairly obvious what the code does from the section on method interception, with the exception of the *set* pointcut primitive. For field interception, we have *set* and *get*, which refer to the assignment to a field and the read of a field respectively. The argument passed into the advice is the new value that the field will be set to.

3.1.6 Aspect precedence

If an object has several aspects that give it advice, there is sometimes a need to explicitly order the aspects. Otherwise they are ran in the order that the interpreter reads them (this is in line with other popular AOP suites).

To give an ordering to the aspects, we include the precedence syntax for it (although with hindsight, this syntax appears a bit superfluous and it should be made more succinct):

```
module Aspects{

  aspect precedence{
    precedence: Aspect1, Aspect2, Aspect3;
  }

  aspect Aspect2{...}
  aspect Aspect1{...}
  aspect Aspect3{...}

}
```

The above precedence syntax will make sure that during the weaving process, Aspect1 will be the first to be added to the ad visors of a joinpoint, then Aspect2 and lastly Aspect3.

This wasn't a particularly interesting part of the project, so not much further work was done on it.

3.2 OOP and AOP unification in AOD-Thorn

3.2.1 Motivation

In most AOP systems, there is a disparity between aspects and normal classes. Firstly, in language extensions such as AspectJ, aspects are generally an extension of classes, in that they can have methods, fields, as well as joinpoint / pointcut / advice definitions. Secondly, in AOP systems implementing AOP as a library, aspects tend to have advice defined in the target language, and pointcuts / joinpoints defined in some external language or DSL [5].

All of these systems can define advice that can refer to objects / classes (and it would be very silly if they couldn't), and aspects can interact with objects in all the ways that a class can (instantiate, access a field, call a method, etc). However, the same does not apply the other way round.

Most significantly, there exists a class of languages, lets call it class θ, which includes most of the popular Java ones, where aspects can instantiate objects but objects have no control over the instantiation of aspects, the AOP system controls aspect instantiation.

It has been argued that this is in general an undesirable asymmetry[40], as some scenarios can be made more complicated needlessly.

3.2.1.1 Sidenote: Classpects by Hridesh Rajan et al.

While work on this part of the project started as an idea by the author, it later turned out that a treatment of the idea had already been applied to the .NET platform by Hridesh Rajan and Kevin Sullivan, with a construct called a classpect, which modifies definitions of classes include pointcuts and advice (and therefore removes the distinction between an aspect and a class) and allows programmer controlled aspect instantiation. The authors of classpects have the following claims for their work:

"First, we can realise a unified design without significantly compromising the expressiveness of current aspect languages. Second, such a design improves the conceptual integrity of the programming model. Third, it significantly improves the compositionality of aspect modules, expanding the program design space from the two-layered model of AspectJ-like languages to include hierarchical structures."[39]

3.2.1.2 Problems with the θ class of languages

The θclass of languages can cause problems for certain applications.

[5]for example in JBossAOP we use annotations / XML for joinpoints / pointcuts, and advice is defined in Java

Firstly, since the θ aspects cannot be instantiated by classes, there exists no path between the standard program entry point (... main(String args[]) in Java) and code which can instantiate θ aspects. Given this, the technique used to instantiate aspects is to have the programming environment do it (we look at how AspectJ instantiates aspects in section 3.2.1.3), instead of the programmer.

Section A.1 in the appendix describes an experiment that shows the aspect instantiation behaviour of aspects in AspectJ. We can see from this that something akin to lazy instantiation is utilised by AspectJ for aspect instantiation. This kind of instantiation mechanism can make reasoning about correctness harder, for example in the the case the constructor has side effects. In AspectJ, there is a restriction that an aspect may only use the default constructor, because it is called by the framework which does not have access to objects (which would be used as arguments to a non-default constructor) outside of what the framework uses.

More importantly than the constructor issue is how to reference an aspect from an object. Many θ languages (including AspectJ) allow objects to *reference* (not construct) aspect instances, but since objects cannot instantiate aspects, aspect instances need to *inject* references to themselves into objects which need to interact with them. This isn't the type of thing one expects to need to do in order to interact with an aspect instance, as it seems long winded and unnecessary. Not only is it long winded, but it requires that Aspects know about the objects that will need them. This asymmetry can lead to unnatural designs.

On the other hand, one may argue that dependency injection (i.e. Spring framework) is a popular trend in the OO world, so aspects having to inject references to themselves into objects isn't such a bad idea. However, building a DI framework for Thorn that could handle injecting aspects would require that the DI framework can instantiate the aspects, which would again be difficult in the θ class of languages.

3.2.1.3 Instantiation models in AspectJ

Apart from the singleton instantiation model, AspectJ (as well as many other AOP frameworks such as JBossAOP) supports four more instantiation models. They are described as follows:

perthis According to the AspectJ documentation, a perthis model is defined as follows: "If an aspect A is defined perthis(Pointcut), then one object of type A is created for every object that is the executing object (i.e., "this") at any of the join points picked out by Pointcut. The advice defined in A will run only at a join point where the currently executing object has been associated with an instance of A."[45] In other words, if a class β contains a joinpoint defined in a pointcut in the aspect α which has the perthis instantiation model, then every instantiation of β will cause an instantiation of α which is bound only to the instance of β which caused α to be instantiated.

pertarget "Similarly, if an aspect A is defined pertarget(Pointcut), then one object of type A is created for every object that is the target object of the join points picked out by Pointcut. The advice defined in A will run only at a join point where the target object has been associated with an instance of A."[45] In other words, if an object of class β performs an operation on an object of class δ , and the operation performed by β is a joinpoint defined in the pointcut in the aspect α, then an instance of α is created whenever an object of β performs the joinpoint and this instance is bound to the δ object which is the target of the operation.

percflow percflowbelow "If an aspect A is defined percflow(Pointcut) or percflowbelow(Pointcut), then one object of type A is created for each flow of control of the join points picked out by Pointcut, either as the flow of control is entered, or below the flow of control, respectively. The advice defined in A may run at any join point in or under that control flow. During each such flow of control, the static method A.aspectOf() will return an object of type A. An instance of the aspect is created upon entry into each such control flow"[45]In other words, this is a dynamic pointcut, in the sense that we need to look at what has taken place in the program (by i.e. checking the stack) before we decide whether the pointcut has been matched.

These are sometimes seen as conceptually complicated (OO programmers are used to , and cannot implement some functionality in a straightforward manner (i.e. singleton is too coarse, perthis / pertarget / percflow is too granular). If for example, we have a group A of objects of class C, and group B of objects of class C, these instantiation models do not support an aspect instance per group very intuitively. Further, turning an aspect on / off for an object/group is also relatively difficult, as described in section 3.2.3.

3.2.2 Example scenario - why we need referenceable aspects + user instantiable aspects

Lets say we have a webserver, which we wish to profile when it experiences high loads (i.e. we wish to find out which parts of the server to optimise when we experience real high loads). We can clearly do this using aspects, profiling is a typical application of aspect oriented programming. But how do we *only* profile when experiencing high loads? Lets say we have an object α that knows about the load being experienced by the server, i.e. its some kind of monitoring object, and our language had a restriction that objects could not reference aspect instances. This could be achieved by advice to profiled joinpoints first asking α about the load and whether the load is high, and then changing the control flow of the profiling advice accordingly. This appears a somewhat contrived solution to the problem.

Firstly, does it really make sense to ask α on each time the joinpoint is reached? The predicate check could be expensive (it will always be more than zero, also the AOD-Thorn interpreter is slow!), and while the expense could be reduced (i.e. caching answers), the programmer will still have the burden of adding this predicate check to each advice in the aspect, introducing scattering.

Secondly, does it really express what the programmer wants / encourage good design? If the application usually takes some sort of action whenever it starts experiencing high loads (for example, emailing an administrator), the set of such actions are likely to be made in a location referenced by α. For example, an object β which is referenced from α may have a method like the following:

```
...
 highLoadExperienced(){
    emailAdministrator();
    //other actions to take when
    //high loads experienced
 }
...
```

Then, with non referenceable aspects, we β may take one set of actions, while the aspect instance would take another set (i.e. start profiling). Clearly, it might be neater if β could have code like this:

```
...
 highLoadExperienced(){
    emailAdministrator();
    profilingAspect.startProfiling();
    //other actions to take when
    //high loads experienced
 }
...
```

There is a clear need for core code to be able to reference aspect instances, so this functionality was included in AOD-Thorn, much as it is included in other AOP frameworks.

The process for the core code to obtain a reference to an aspect instance should be made as easy as possible, hence the addition of user instantiable aspects in AOD-Thorn.

3.2.3 Selective vs unselective

AOD-Thorn aspect instances can be selective or unselective; we discuss the differences.

Most θ languages have a singleton aspect instance by default (with extra mechanisms for instantiation, such as perthis or pertarget), with an implicit reference to the aspect instance when executing advice.

The case of having a singleton aspect usually lends itself to the concept of unselective aspects, in that the advice is applied to all instances of the types that contain the joinpoints, and therefore the advice is executed during the execution of joinpoints in these types.

For many cases, especially where the aspect has no state, this is an adequate mechanism. However, it is often inconvenient to have a singleton instance of an aspect which applies to all instances of the joinpoint containing type, as seen in the example scenario . Where aspect instances have a state, we may wish to give different advice to different sets of objects of the same type, based on the aspect instance state. Using unselective aspect instances, we can achieve this by using a table method. For example, we can have two tables of form:

$$\mathit{ObjectLookup} : \mathit{Object} \rightarrow \mathit{SetID}$$

$$\mathit{FieldLookup} : \mathit{SetID} \rightarrow \{\mathit{Field}\}$$

A field is of form $field = \mathit{fieldId} \times \mathit{type} \times \mathit{value}$.

Then, given an object O, we can retrieve the value of a field $f \in \{\mathit{Field}\}$ for O by applying the two table lookups, i.e. $(\mathit{FieldLookup}(\mathit{ObjectLookup}(O))[f] \downarrow_3 (\downarrow_3$ gets the value, i.e. the 3rd value of the tuple).

. It is easy to see why this is seen as inconvenient.

The situation is simplified if we only have a requirement that the advice is applied to a subset of all the instances of a type.

3.2.3.1 Definitions (simplified, informal)

ID is overloaded

$$\begin{array}{lcl}
\mathit{pointcut}\,\pi & = & \{\mathit{joinpointRef}\} \\
\mathit{joinpointRef}\,\mu & = & \mathit{methId} \cup \mathit{fieldId} \\
\mathit{method} & = & \mathit{methodId} \times \mathit{type} \times \mathit{arity} \times \mathit{methBody} \\
\mathit{field} & = & \mathit{fieldId} \times \mathit{type} \times \mathit{value} \\
\mathit{joinpoint} & = & \mathit{method} \cup \mathit{field} \\
\mathit{joinpoints} & = & \{\mathit{joinpoint} \,|\, \mathit{joinpoint} \in \mathit{Program}\} \\
ID & : & \mathit{methId} \rightarrow \mathit{method} \\
ID & : & \mathit{fieldId} \rightarrow \mathit{field}
\end{array}$$

We refer to a class C as C and the set of instances of C in the program at some point in time as $\mathit{inst}(C)$. A pointcut π is a member of pointcuts Π defined in an aspect. Every advice has one pointcut and every pointcut has one advice. An aspect includes a set of <pointcut, advice> pairs, i.e. $aspect = (..\times(\pi\times advice))$.

Then, if we have the requirement that a set of advices with pointcuts Π are only given to $O : O \subseteq \bigcup\{inst(c)|\mu \in \pi \wedge \pi \in \Pi \wedge ID(\mu) \in c\}$(i.e. some subset of objects which are instances of classes which contain the joinpoints referenced in the pointcuts of the advices, in other words, we wish to apply the advice only to some objects that match the pointcuts of the given aspect), we have a simpler solution.

The first step for an advice only applied to O will be to check if the object o under consideration of being given advice to is $\in O$ (O would be a set held in the aspect instance), and if it isn't, to abort the execution of the advice.

So in other words, we could have something like the following (in pseudocode):

```
aspect asp{
  Set<Object> advisedObjects

  pointcut {
     if(joinpoint in advisedObjects)
       give advice
  }
}
```

However, this approach has some major faults:

1. It requires the programmer to to explicitly code this functionality into their aspects
2. All advices must have the $o \in O$ check, resulting in code scattering, against the principles of AOP. We need to either live with it, or give advice to advice, increasing the conceptual burden on the programmer.
3. There is always an advice invocation, even if an object will not be advised. Depending on the implementation of the AOP system, this may be relatively expensive (i.e. some AOP systems which create a large number of objects on each advice invocation).
4. In the θ languages, either the aspect instance has to inject a reference to itself into objects so that objects may tell the aspect instance which objects to bind to (which is somewhat long winded), or the aspect instance will have to be what creates the advised objects (which can be very confusing).

Clearly, table / set based methods like this leave a lot to be desired. While we have perthis / pertarget, these may spawn too many aspect instances. We may wish to bind an aspect instance to a single object which whose class contains the joinpoints in the aspects pointcut but not bind an aspect instance to each instance of the class that contains joinpoints defined in the pointcut (as would happen with perthis / pertarget), or, we may wish to bind an aspect instance to a group of objects.

Selective aspects can help us here.

Selective aspects A selective aspect instance, specified by adding the *selective* keyword after the *aspect* keyword, will declare that the aspect instance will only advise objects added to it using the *addToAsp(ObjectTh, AspectObject)* function, which is equivalent to switching an aspect on for an object at runtime (switching it off at runtime is equally simple, simply use *delFromAsp(ObjectTh, AspectObject)*). This removes the need for the programmer to explicitly build this into their aspect, reducing the need for each advice to have to check every time it is ran.

Further, as execution of Thorn code is slower than execution of Java code, moving this check into the interpreter should result in faster joinpoint evaluation, as the execution of a joinpoint is not slowed down as much as if Thorn code was used for the inclusion check (the implementation chapter has more details on this).

Finally, it removes the scattering inside the advices, as there is no need for each piece of advice inside the selective aspect to check whether the object is to be advised.

Of course, selective aspects have problems of their own:

1. something needs to pass it the objects that the selective aspect instance is to bind to, which may decrease modularity as there must be a reference to the aspect instance. The core code may not be completely oblivious to the aspects solving the cross-cutting concerns, however, aspects still remain useful.

2. as selective aspects can only bind to already constructed objects, they cannot give advice to an objects during the initialisation phase (i.e. to constructors). This should not be much of an issue. Simply keep the advice to constructors in an unselective aspect. Proper tooling should downplay fears of spreading the advice to a class across several aspects (and thus making it harder for the developer to work out / understand program flow).

3.3 Transparently distributed aspects

Local aspect-oriented programming is relatively straight forward. We insert *advice* at joinpoints which match predicates (pointcuts) defined in aspects.

In languages with shared state, giving advice to code objects several threads is not much more difficult than if the advice takers were in the same thread, synchronisation mechanisms can be used. An aspect instance can live in one thread and give advice to objects in several different threads, and this works correctly.

However, the case of stateful aspect instances (when advice has side effects on the aspect instance, i.e. reads and modifies fields of the aspect instance) is more difficult in Thorn.

Since Thorn is a language designed to make it easy to write concurrent as well as distributed applications as easy as possible, it makes sense to make aspect-oriented programming in Thorn in the context of distributed / concurrent code as easy as possible.

Traditionally concurrent code Thorn has no shared state, concurrent code is ran on separate JVM instances (each component in Thorn is ran on a separate VM), message passing is used instead of synchronisation / locking. This makes it difficult to give advice to concurrent code, since the Aspect instance can only have a reference to the objects running on the same VM. This is made worse, when for example, a pointcut applies to multiple objects (on separate VMs) and the aspect is stateful.

Traditionally distributed code Furthermore, it may be a requirement to advise objects on separate VMs which do not interact with each other (i.e. embarrassingly parallel), but the advice has to be aggregated somehow. For example, there may be several webservers serving a static web page, however, we have the need to count how many times the page has been accessed across the whole distributed system. A trivial task if there is only one webserver (i.e. just insert a before advice which logs invocations of the serve HTTP request routine), however this task is less trivial if aggregation of this log data is required across all the servers, we may need to write networking code in our aspects, so that they communicate with each other and come up with a single number.

Part of the philosophy of Thorn is script it now and, if required, evolve it later. As such, the author proposed that aspecting distributed code should have a mechanism of being done very easily, even if it isn't as efficient as if the programmer exerted more control over how the aggregation happens.

Lets look at a motivating scenario.

3.3.1 Motivating scenario

Spec: To create a client / server application. The client asks two servers, one in America and the other in Britain, for the news of the day. The servers read the file from a local file in order to serve the request and we wish to profile the average time taken across all the news servers for the servers to read the news file, process it and then send the result to the client[6].

The application started with a single server in the USA but has moved to an experimental multiserver architecture, so now it has the British server also.

One way to do this, would be to to create a log server, which receives the time taken logs from the aspect instances of the various news services. However, this places a burden on the programmer to modify the profiling aspect that was written when the application had a single server architecture, and since the multiserver architecture is experimental, the distribution aware aspect code may have to later be reverted.

If we have transparently distributed aspects, we can for example, have the client initiate an instance of the profiling aspect as the master, and the aspects running on the news service servers as aspect slaves. Then, all the aspect state and side effects (i.e. IO operations) are done at the master.

This means, for the following NewsClient (following multiserver architecture) and NewsService, we can keep the original aspect code (ProfilingAspect) written for a single server architecture without modification .

Listing 3.1: NewsClient.th

```
component NewsClient{
  body {
   // get the handles for the two server components
   americanNewsSvc = site("thorn://newscorpusa.com:1234");
   britishNewsSvc = site("thorn://newscorpuk.co.uk:1234");

   // invoke the getTheNews() method on the server
   americanNews = americanNewsSvc <-> getTheNews();
   britishNews = britishNewsSvc <-> getTheNews();

   println("American news:\n" + americanNews);
   println("British news:\n" + britishNews);
  }
}NewsClient;
```

Listing 3.2: NewsService.th

```
component NewsService{

  sync getTheNews(){
    readNewsFine(); //implementation ommited
  }

```

[6]we can't do this performance measurement on the client, due to possible differences in network lag between the two servers, they are in different countries!

```
  body{
   while(true) serve;
  }

}NewsService;
```

Listing 3.3: ProfilingAspect.th

```
aspect unselective ProfilingAspect{
  var counter := 0;
  def incCounter()={counter := counter+1;}

  around: exec jp[getTheNews] called(){
    // nanosecond precise system clock value
    timeBefore = nanoTime();
    // proceed with the getTheNews computation.
    proceed();
    timeAfter = nanoTime();

    // logs the time taken somewhere locally ,
    // e.g. in a txt file on the hard drive
    log(timeAfter - timeBefore);

    azpct.incCounter();
  }
}
```

As mentioned previously, if the language did not have transparently distributed aspects, we would need to modify the profiling aspect with code that dealt with the coordination between a log aggregator service and the aspects give advice to the NewsClient instances.

For example, we may need code like this:

```

//single server architecture:
around: [getTheNews] called(){
  ...
  log(timeAfter - timeBefore);
  ...
}

//multie server architecture:
remoteTimeTakenService =
    site("thorn://logAggregator.com:servicePort");

around: [getTheNews] called(){
  ...
  // logRemotely(timeTaken)
  remoteTimeTakenService <->
      logRemotely(timeAfter - timeBefore);

  remoteTimeTakenService <-> incCount();
  ...
}

component RemoteTimeTakenService{
  val counter := 0;

  sync incCount() = {counter := counter+1;}

  sync logRemotely(timeTaken){
    log(timeTaken);
  }

  body{while(true) serve;};
}
```

As we can see from the example, there is work involved from the programmer to get the aspecting to work in a distributed setting. Infact, as components in Thorn run on different JVM instances, the programmer has to put this work in, even if those components are running on the same machine (although this does not apply to our scenario, as its purely a distributed application rather than a concurrent one).

Furthermore, the reusability of the Aspect code is reduced, it is now polluted with networking code.

To achieve the transparently distributed aspects, we can have an aspect running as the master. The master holds the state of the aspect instance as well as the aspect AST, while the slaves hold the aspect AST and a reference to the master.

To create a master aspect instance, we have the masterAspect built in function, which takes two arguments:

1. Instance of an aspect (selective or unselective)
2. A name to use for the service (RMI registry is used, more on this in the implementation chapter)

There is further a slaveAspect built in function, which also takes two arguments:

1. Hostname of the RMI registry which has the master aspect
2. The name of the master aspect instance

In our scenario, the client would use the following code during startup:

```
program entry point ...
// instantiated as before
profilingAspect = ProfilingAspect();

masterAspect(profilingAspect ,
             "masterProfilingAspect");
...


aspect unselective ProfilingAspect...
// doesn't change, as before
```

and on the slave machines (the ones with news service):

```
program entry point...
slaveAspect("masterhost",
             "masterProfilingAspect");

aspect ProfilingAspect...
// doesn't change
```

Infact, the slaveAspect returns an aspect instance (which as explained in the implementation section) is a wrapper around a normal aspect instance. As such, it also works with selective aspects, as we shall see next.

3.3.2 Another motivating example, with selective aspects

Lets take a look at another example. We demonstrate the idea of selective aspects, as well as distributed aspects. This is not an example of good design (traditionally, aspects should not have core concerns such as raising an alarm, although the bellow may be seen as an instance of the observer pattern), however it does show off the features of transparently distributed aspects.

The scenario: we have two buildings, one which contains several temperature sensors connected to a network connected computer as well as an alarm system, and the central building which receives readings from these sensors.

The sensors code may look like this. They can sense a temperature, in which case they log it locally (in this case just through println), but they are also able to activate the alarm, with a parameter to the alarm, in this case a message.

Listing 3.4: sensors.th

```
module Sensors{

  class TempSensor(name){
    def sense(temperature) =
      {println("Sensed: " + temperature);};

    def activateAlarm(message) =
      {println("Alarm! Instructions: " + message);};
  }
}
```

We started with an architecture where the sensor knew when to invoke an alarm, but now the central server knows the temperatures that should be experienced at the moment (based on time of year, temperature outside, and so on), so the slave buildings must ask for it all the time. We do not want to modify the TempSensor class to keep it simple, certainly not with any networking code.

In our single computer architecture, we used to have an aspect that intercepted the sense method (very much in the style of the observer pattern), and based on the temperature sensed, it would react accordingly:

```
module Aspects{

 aspect selective SensorLog{

    // keep a count of how many alarms have been activated
    var alarmsActivated := 0;
    def getAlarmCount() = {return alarmsActived;};
  def incAlarms() = {alarmsActivated := alarmsActivated +1;};

    // very simple logging system
    def log(msg) = println(msg);

    def getAlarmThreshold() = {return 5;};
    def getEmergencyThreshold() = {return 9;};

    before: exec jp[(sense)] args(arg:int){
      azpct.log(this.name + " sensed: " + arg);

      if(arg > azpct.getAlarmThreshold()
          && arg <= azpct.getEmergencyThreshold()){
        // we should raise an alarm
        // but there is no need to panic
        this.activateAlarm("Notify authorities");
        azpct.incAlarms();
      } else if(arg >= azpct.getEmergencyThreshold()){
        // everyone needs to leave the building!
        this.activateAlarm("Evacuate immediately!");
    azpct.incAlarms();
      }
```

```
        }
}}
```

This aspect worked in the single machine scenario, but it also works in the distributed scenario when we use transparently distributed aspects.

We have decided that we wish to partition the sensors in the slave buildings into two groups, locationA and locationB. All we need to do now is to create two aspect instances on the master machine, and export them on an RMI registry (details in the implementation chapter) using the masterAspect built in function, giving it a name, i.e. “locALog”.

Listing 3.5: mastersite.th

```
import Aspects.*;

locationALog = SensorLog();
locationBLog = SensorLog();

masterAspect(locationALog, "locALog");
masterAspect(locationBLog, "locBLog");

println("Master running");
```

Now, lets look at the code that would be written on the site with the temperature sensors. We simply create slave aspects (aspect AST lives locally, but the aspect we “instantiate” is a proxy for the remote aspect), using the slaveAspect method, to which we supply a connection string, e.g. localhost on the default RMI port (1099) using “localhost”, and the name of the master aspect that we wish to create a proxy for.

Then, since this is a selective aspect, we must add the sensor objects to the aspect instances, and we use the *addToAsp* built in function for this. Sensor 1a and 1b belong to one group of sensors, i.e. they’re in room 1 of the building, and sensor 2 is another room.

Listing 3.6: sensorSite.th

```
import Sensors.*;
import Aspects.*;


tempSensor1a = TempSensor("Sensor 1a");
tempSensor1b = TempSensor("Sensor 1b");

tempSensor2 = TempSensor("Sensor 2");

slaveA = slaveAspect("localhost", "locALog");
slaveB = slaveAspect("localhost", "locBLog");
```

```

addToAsp(tempSensor1a, slaveA);
addToAsp(tempSensor1b, slaveA);

addToAsp(tempSensor2, slaveB);
```

Then, if we executed the following code (which simulates the temp sensors sensing temperatures and at the end of it asking the aspects how many alarms were raised) :

```
tempSensor1a.sense(6);
tempSensor1b.sense(9);

tempSensor2.sense(6);

println("Num of alarms at siteA: "
        + slaveA.getAlarmCount());

println("Num of alarms at siteB: "
        + slaveB.getAlarmCount());
```

we would get the output:

```
Sensor 1a sensed: 6
Notify authorities

Sensor 1b sensed: 9
Evacuate immediately!

Sensor 2 sensed: 6
Notify authorities

Num of alarms at siteA: 2
Num of alarms at siteB: 1
```

3.3.2.1 What the above example shows

It should be apparent that the above example has shown the ability of transparently distributed aspects to do a few things:

1. There was no need to add any networking code - making the code more concise, and further there was no need for the programmer to do anything when they moved to a distributed system, other than use masterAspect and slaveAspect, instead of instantiating as usual

2. It is possible to have selective aspects that can bind to objects on different computers - i.e. when using addToAsp, it was possible to have added tempSensor1a and tempSensor1b from different computers to the master aspect instance, the output for num of alarms at siteA would still have been the same

3. It is possible to send arguments and receive return values from the master aspect in order for the slave to give advice to the joinpoint it has intercepted

How all of this works is explained in section 4.3.

Chapter 4

Implementation

The implementation section should shed light on how the new features are implemented in the interpreter. The implementation documentation for the features which the author felt are more mature / likely to make to make it into AOP suite for Thorn is written in far more detail than the features which were more experimental and the author felt they were not particularly likely to be useful (e.g. distributed aspects).

4.1 Vanilla aspecting

An aspect in AOD-Thorn is an extension of a class. It can do everything that a class can do, i.e. be instantiated using different constructors, have methods and fields, but it also has advice and pointcuts.

4.1.1 Syntax / AST changes

Syntax is presented in EBNF form. The definitions of the quantifiers are as follows: + is 1 or more, * is 0 or more, ? is 0 or 1. | refers to a choice. Identifiers starting with a capital letter refer to other rules in the syntax, although they may not be fully defined the following syntax description. Some details were omitted, for example ANY_REGULAR_EXPRESSION, or syntax previously present in Thorn, i.e. Method. The implementation can be seen in the javacc grammar definition file.

```
Statement ::= AS_BEFORE| AspectDecl

AspectDecl ::= 'aspect', ('selective'|'unselective'), Name,
                    ClassFormals, Purity /* a thorn concept used for serialisability
of a class*/,
                     '{', PrecedenceMembers?,
                     AspectMembers, '}' ,';'

PrecedenceMembers ::= 'precedence', Id, (',', Id)*, ';'

AspectMembers ::= ClassMembers | AdviceDeclaration

AdviceDeclaration ::= AdviceLocation, ':', PointcutPrimitive,
                         Pointcut, Method

AdviceLocation ::= 'before' | 'after' | 'around'

PointcutPrimitive ::= 'get' | 'set' | 'exec'

Pointcut ::= Joinpoint+

Joinpoint::='jp[', ANY_REGULAR_EXPRESSION, ']'

VarExpressions ::= 'this' | Id | 'azpct' | 'origin' | 'joinpoint()' | 'proceed()'
```

On top of the syntax changes, there needed to be changes made to the AST. The AST is defined in a Python file (as described in 2.3.1.1), which generates Java classes that represent the AST nodes within it.

The parser generated by the JavaCC grammar definition file tries to match a rule, and if it does, it creates an instance of the AST node associated with the rule.

For example, the following is part of the changes made to the Python AST file. Comments about how this file works are provided:

Listing 4.1: ast-fisher.py

```
...

//definition of the AST node describing a pointcut
AspectPointcut =
        // represented by a class called Pointcut
        // which derives from the class Syntax
        CLS("Pointcut", Syntax,
        // a pointcut has a list of joinpoitns
        [LIST("joinpoint", "joinpoints",
        // the AST subtrees representing a joinpoint
```

```
          // are of type AspectJoinpoint
          AspectJoinpoint)],
          // this isn't particularly important for now,
          // its a flag which indicates whether we want
          // to make it possible to deepy copy this AST tree
          isDeepCopyable=False)

// The next follow the same format as the previous

AspectJoinpoint = CLS("Joinpoint", Syntax,
                   [FIELD("pointcut", String,
                   isChild=False)])

AdvicePtctPrimitiv =
                  CLS("AdvicePtctPrimitiv", Syntax,
                   [FIELD("pointcutPrimitive",
                   AdvicePtctPrimitiveEnum,
                   isChild=False)])
...
```

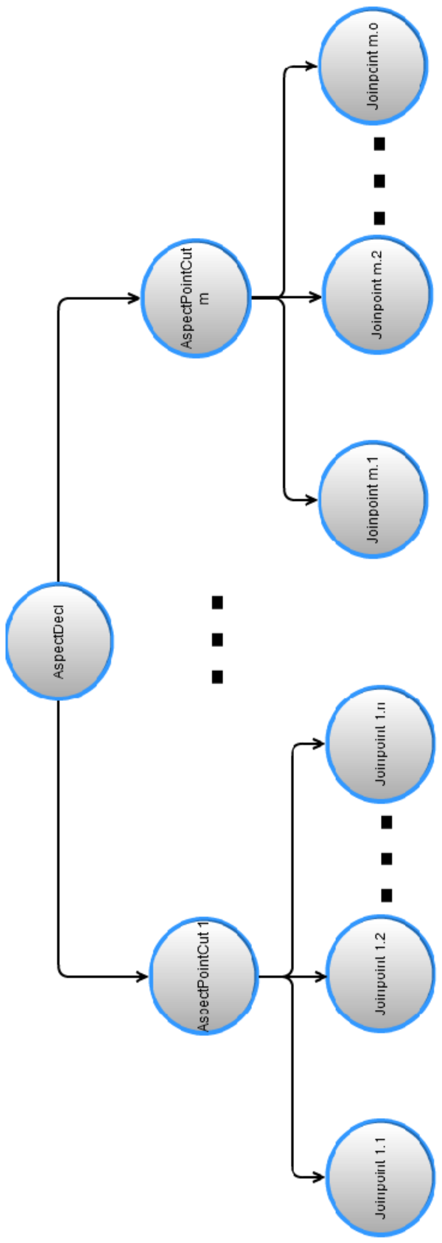
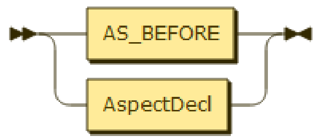
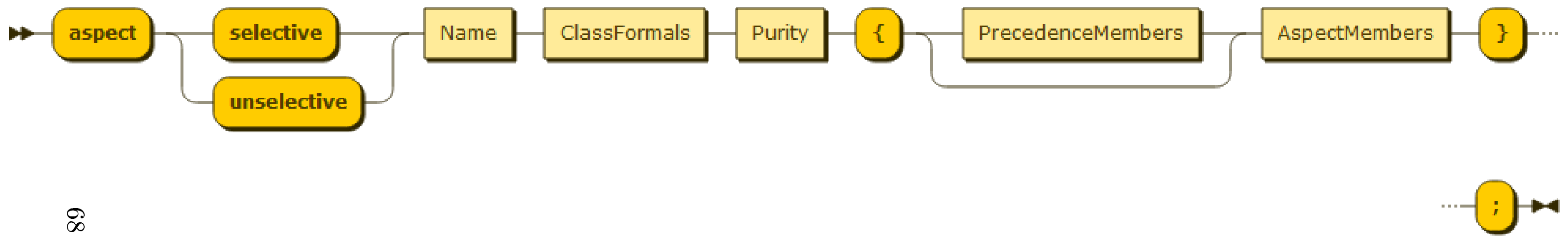


Figure 4.1: Example: Aspect AST tree. 1:N cardinality between aspects and pointcuts, and 1:N cardinality between pointcuts and joinpoints. Note that AspectDecl trees and Pointcut trees can also have other types of subtrees, i.e. method definitions in an aspect.

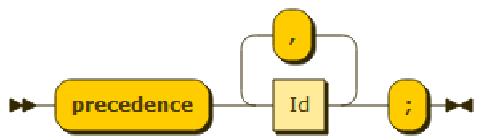
Statement:



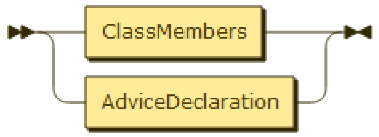
AspectDecl:



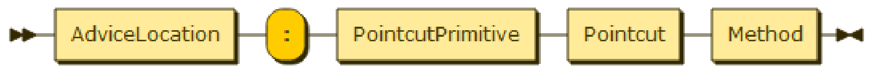
PrecedenceMembers:



AspectMembers:



AdviceDeclaration:



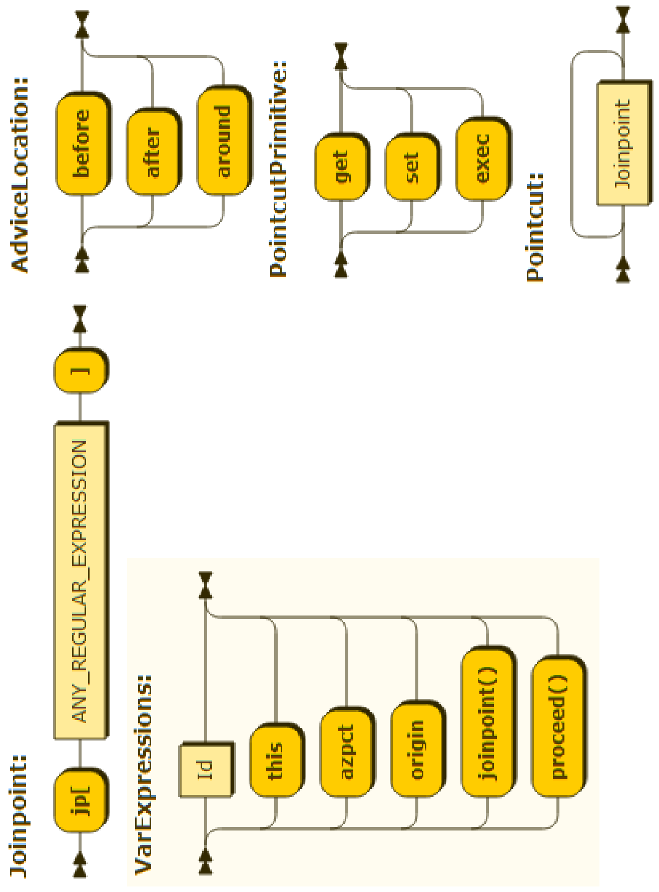


Figure 4.2: Railroad diagrams

4.1.1.1 Preparing the interpreter for giving advice

In general, whenever there is a change in the AST / Syntax, changes need to be made to the AST visitors.

Sealant As mentioned before, the sealant is what is used to give seals (see 2.3.1.2) to the AST nodes. Furthermore, it is responsible for modifying and assigning *Environments.*

Environment An environment[1] in the Thorn interpreter is a further mechanism for scoping. An environment is essentially a recursive data structure mapping identifiers to their values. For example, a method m(arg1,arg2) will have an environment which maps the identifier arg1 and arg2 to the parameters passed into the method, and furthermore, it will point to the environment of the parent of the method, i.e. an object (the environment for which will map fields of the objects to their values).

The environment is then used for value lookups when evaluating an AST tree. For example:

```
module CONSTANTS{
  val PI = 3.142;
  val EXP = 2.718;
}

class Bernoulli {
import CONSTANTS.*;

val name = "Jacob Bernoulli";

fun introduce_self(shortIntroduction) {
  if(shortIntroduction){
    println("Name: Bernoulli. My invention: e");
  }
  else {
    println("My name is " + name +
          ". I discovered the value of e = "
          + EXP );
  }
}
}
```

When executing the *introduce_self(bool)* method, the interpreter will need to consult the environment of the method and the environment of the class. For

[1]class fisher.statics.Env

example, the snippet *if(shortIntroduction)* requires the interpreter to first look into the environment attached to the method *introduce_self()* the value of the identifier *shortIntroduction.* The sealant would've given this identifier a seal, which is used as a key into the table mapping identifiers (or more precisely, the seals of identifiers) to values in environments. Therefore, the environment of the most precise scope that *if(shortIntroduction)* exists in is consulted for the value. Since the value exists there, the value is found and the if statement can be evaluated.

For the snippet *"My name is " + name,* the interpreter asks the environment for the value of the *name* identifier. Since this identifier isn't a method argument, nor is it defined inside the method, the environment asks its **parent environment** for the value of *name.* The parent environment belongs to the instance of the Bernoulli class. The parent finds it, and returns the value to the child.

The snippet *"... e = " + EXP* needs an additional step. Since EXP identifier is neither defined in the environment of the method, the environment of the class is consulted. However, the *EXP* identifier is not defined in the Bernoulli class. However, notice that a module is not something that is instantiable, so there is no need for it to have an environment instance, in which case, there is no need for for the class environment to ask its parent environment (it doesn't have one).

Note also that we can be selective about the parts of a module which we want to be referenceable, i.e. we may only want to import CONSTANTS.PI to be use able by the Bernoulli class. Therefore, it is incorrect to simply refer the class environment to some mechanism where the identifiers in a module are defined. Instead, the environment of the class (statically, i.e. common to all instances of the class) contains references to only imported identifiers from the module.

This means that to get the *EXP in "... e = " + EXP,* the class environment first checks through its map of identifiers which are defined in the class, and if its not found there, it will check through a map that contains identifiers that have been imported from other modules. Therefore, the class environment holds identifier values from both the class as well as the module.

Required changes It was required that the sealant is made fully aware of all the new AST nodes that were introduced as part of the introduction of aspects. For example, there needed to be seals assigned to all members of an aspect (fields, advice, etc), as well as the new keywords such as *azpct* or *origin.* As aspects closely resemble classes (they are extensions), the same approach towards sealing was taken for common parts of classes and aspects.

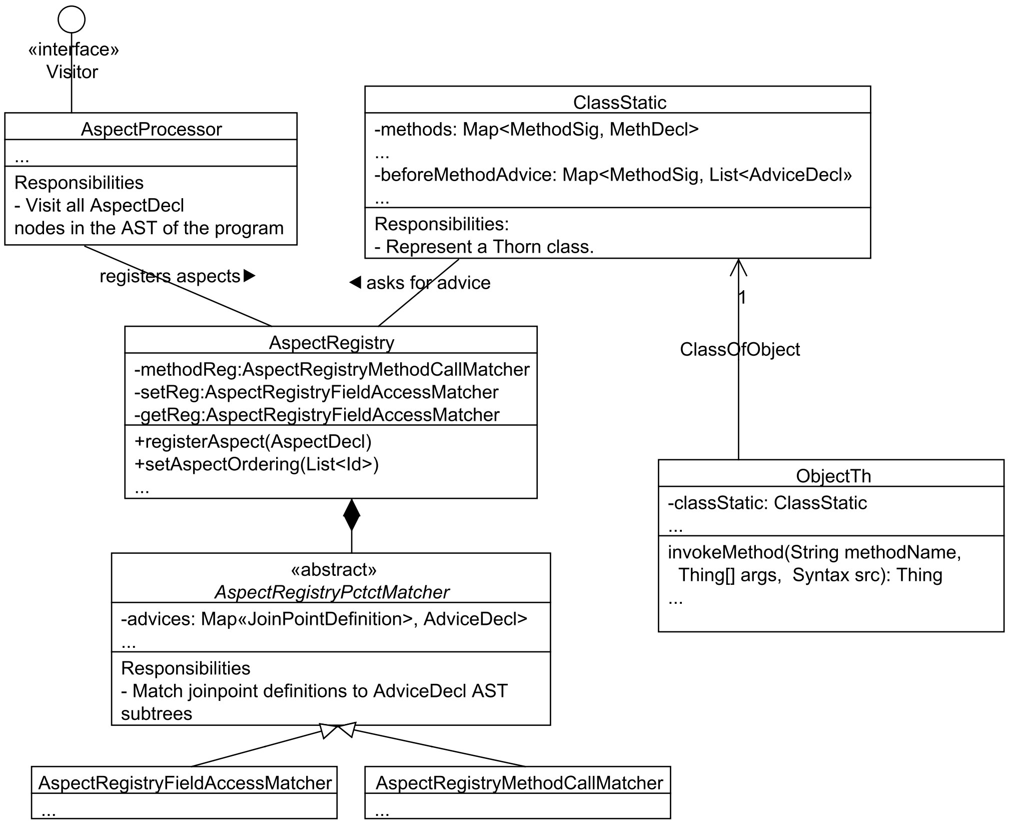

«interface»
Visitor
AspectProcessor
...
Responsibilities
- Visit all AspectDecl
nodes in the AST of the program
ClassStatic
-methods: Map<MethodSig, MethDecl>
...
-beforeMethodAdvice: Map<MethodSig, List<AdviceDecl»
...
Responsibilities:
- Represent a Thorn class.
registers aspects ▶
◀ asks for advice
1
ClassOfObject
AspectRegistry
-methodReg:AspectRegistryMethodCallMatcher
-setReg:AspectRegistryFieldAccessMatcher
-getReg:AspectRegistryFieldAccessMatcher
+registerAspect(AspectDecl)
+setAspectOrdering(List<Id>)
...
ObjectTh
-classStatic: ClassStatic
...
invokeMethod(String methodName,
Thing[] args, Syntax src): Thing
...
«abstract»
AspectRegistryPctctMatcher
-advices: Map«JoinPointDefinition>, AdviceDecl>
...
Responsibilities
- Match joinpoint definitions to AdviceDecl AST
subtrees
AspectRegistryFieldAccessMatcher
...
AspectRegistryMethodCallMatcher
...

Figure 4.3: AspectRegistry UML. Unimportant details represented as "..."

4.1.2 AspectRegistry

The AspectRegistry[2] is the main data structure holding information about advice and pointcuts, it is a singleton populated with data at initialisation time.

When the AspectProcessor visitor comes across an aspect declaration in the AST tree (*AspectDecl* in figure 4.1.1), it invokes the *registerAspect(AspectDecl)* method on the AspectRegistry singleton. The AspectRegistry will then go through all the advice declarations (look at AspectMembers in figure 4.1.1) in the aspect declaration, and for each one, will read the joinpoint and save the AdviceDecl AST sub tree in a HashMap that maps from joinpoint definitions (i.e. a [regular expression, number of formals] tuple) to the subtree.

4.1.3 Weaving phase

A ClassStatic is essentially the runtime representation of a Thorn class. For example, the ClassStatic maps from identifiers to methods (a method is stored as an AST).

When a class declaration is visited by the Sealant, a new instance of the ClassStatic is created, which is used by all the instances of the class represented by ClassStatic. When ClassStatic is being constructed, it creates a map mapping method signatures[3] to method AST trees [4]. The constructor is however only given MethDecl trees, so it needs to create MethodSig objects.

The ClassStatic class was enhanced with several maps that map from methods and fields to lists of AdviceDecl trees. Then, when we execute a method or a field access, the Thorn object (ObjectTh) can look at its ClassStatic (each ObjectTh has a ClassStatic bound to it), and use the method signature of the executing method to get a list of advices for the method invocation, or the seal of the field when executing a field access to get a list of advices for the field access. For each type of joinpoint (exec | set | get), there is a list of before, after and around advices that can apply to it, so the original code was modified to take this into account. For example, before doing a method invocation, we first get the before advices for that method, and run them, then we invoke the method, and then we execute the after advices.

Creating the advice maps in ClassStatic is done using the AspectRegistry. The AspectRegistry exposes a set of methods asking for joinpoints and returning a list of advices. For example, the AspectRegistry can take a field seal (the joinpoint definition), and will return a list of AdviceDecl objects (see section 4.1.4 for how this is done).

There is now a general description of how field access advice and method call advice is given. However, this isn't the whole story, and it is continued in section 4.2 when the aspect instantiation mechanism is explored further.

[2]fisher.eval.AspectRegistry
[3]fisher.statics.MethodSig
[4]fisher.syn.MethDecl

4.1.3.1 Field access advice

During the initialisation of ClassStatic, the following actions[5] are taken for fields defined in the class declaration AST tree:

- Create a list of field names (type String)
- Check for collisions, to make sure that the same field has not been defined twice in a class.

This routine was modified, in that after the collision check is performed, we also cache advice for field accesses into the ClassStatic.

For both *get* and *set* (read / write) field accesses, we added maps into the class static, one for each advice location (before, after, around). These map from the field name to AdviceDecl AST trees, which contain the bodies of the advice. So, upon a read[6] / write[7] of an ObjectTh field, the name of the field is used to lookup if there are any advices, and if there are, then they are executed as necessary, before, after or instead of the field access.

4.1.3.2 Method calls

fisher.syn.MethDecl is the AST which contains the statements for a method. We modified the class initialisation routine (the routine is in ClassStatic), so that the routine includes storing a lists of aspects into MethDecl (see figure: 4.4)objects which are used in the class initialisation routine.

Then, during method execution, these lists are used to see which aspects need to be executed before / after / around the main body of the method. Advice statements are executed in the same way as method statements, except for a few small changes (see section 4.2.2 and 4.2.4). More details on this later in the report.

4.1.4 Joinpoint matching

Pointcuts in AOD-Thorn support full regular expressions. For example, we may wish to give advice to every method whose name begins with set and has one argument in a given module / class.

When the Sealant visits an identifier (such as a field declaration in a ClsDecl), it assigns it a Seal object which contains the fully qualified name of that identifier. For example, the method setX in class bar and module foo will have a seal that contains the fully qualified name foo.bar.setX.

[5] fisher.statics.ClassStatic.snagMethodsPatsVarsVals()
[6] fisher.runtime.ObjectTh.LValue(...)
[7] fisher.runtime.ObjectTh.RValue(...)

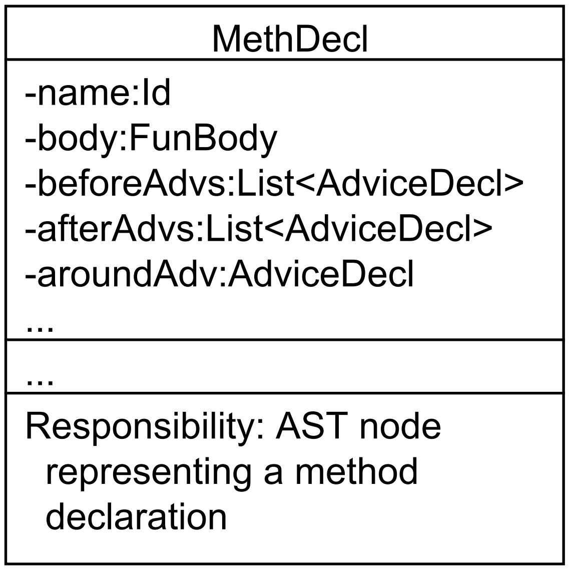

MethDecl
-name:Id
-body:FunBody
-beforeAdvs:List<AdviceDecl>
-afterAdvs:List<AdviceDecl>
-aroundAdv:AdviceDecl
...
...
Responsibility: AST node representing a method declaration

Figure 4.4: MethDecl with Aspects

This allows us to use regular expressions as a relatively rich pointcutting language. For example, if we wish to match all the setters in the foo.bar package, we simply need to use the regular expression *"foo.bar.set\.*"*. Notice that the first two dots were not escaped, whereas the last one was. This is because the implementation simply makes use of the java.util.regex package for regular expressions, and the dot in that package represents any character. Since the dot character is fairly common when defining pointcuts, it is escaped by default, and escaping it in the pointcut is the equivalent of the unescaped dot character. (In hindsight, the wildcard version of the dot could've been replaced by another symbol to avoid the need to escape it)

When the AspectProcessor is visiting AspectDecl AST trees, the regular expressions in the pointcut are compiled, and then stored into a Map<Pattern, AdviceDecl> object in the AspectRegistry. Precompiling the regular expression into a Pattern object means that checking for matches are much faster than if one were to not store the compiled Pattern later. Then, when the ClassStatic is being initialised, the Patterns are matched against the joinpoints, and the advice is stored in the MethDecl AST accordingly

4.1.5 proceed()

When around advice is being given, there is a need for a mechanism to execute the original joinpoint which is being advised. For example, in an example elsewhere in the report, we profile a method by taking a clock reading just before and after the joinpoint is executed.

While this can be achieved using a before and after advice where the aspect instance has a field that stores the clock reading before the joinpoint is instantiated, this is an incorrect approach, especially when the aspect instance applies to multiple objects: even in a non-multithreaded environment, while the advised joinpoint is being executed, another joinpoint may be advised by the same aspect instance with the same advice, overwriting the clock value.

As such, the proceed() function was implemented, as is done in popular AOP suites. Please see the semantics (section 4.2.7.2) to fully understand how it works.

4.1.6 joinpoint()

The joinpoint function will return information about the current joinpoint that is being advised. In Java, this is usually an object from the reflection library. For example, when the equivalent function is called in AspectJ while giving advice to a method, an instance of java.lang.reflect.Method is returned.

Thorn has no equivalent reflection API, so the implementation was somewhat simplified. Currently, it will store the string representation of the seal of the joinpoint. For example, if a method m in class C with arity 0 is the joinpoint, then the joinpoint() function will return a Thorn string with value "C.m\0".

This is done by modifying the semantics of joinpoint execution. Now when an advice for a joinpoint executes, one of the steps is to place a StringTh into the stack frame. The StringTh object is generated from the toString() value of the seal for the joinpoint (recall that all joinpoints have a Seal, and the toString() is cached in the Seal at initialisation time). The execution of advice requires the creation of a StringTh object; this could be made more lazy by only placing the String there, and creating a StringTh if the advice calls joinpoint(), or alternatively other optimisations could be taken (i.e. the advice AST could be processed at initialisation time to check for a call to joinpoint()), however, given the microbenchmark results in the evaluation section, the author found no need to do so.

4.1.6.1 Advice arity and type checking

When advice is given to a joinpoint, sometimes there is a set of arguments that need to be passed to the advice. For example, from our earlier sensors example we have the pointcut:

```
before: exec jp[(sense)] args(arg)
```

Recall that the advice of this pointcut used arg in a String, i.e. used the String representation of arg, so any object with a toString() method would be accepted.

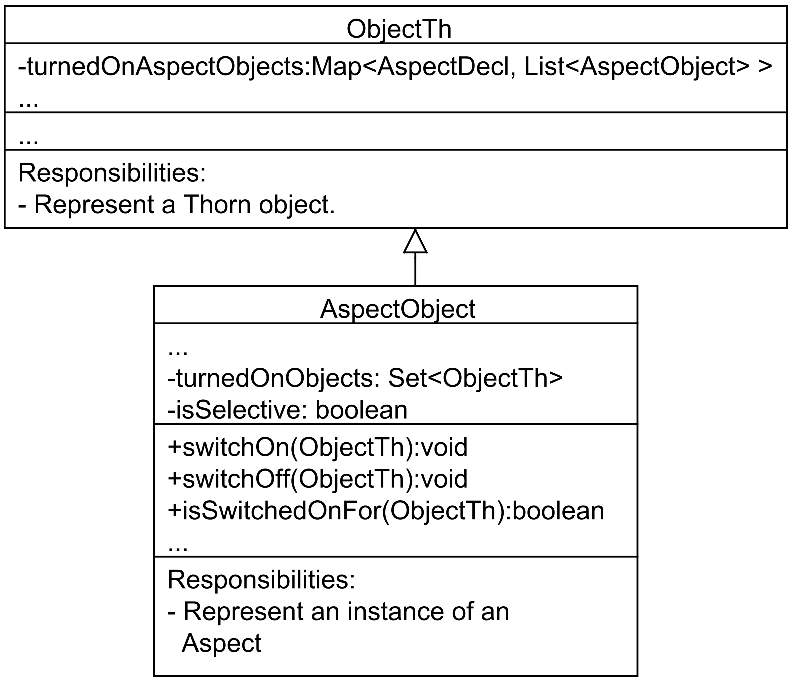

ObjectTh
-turnedOnAspectObjects:Map<AspectDecl, List<AspectObject> >
...
...
Responsibilities:
- Represent a Thorn object.
AspectObject
...
-turnedOnObjects: Set<ObjectTh>
-isSelective: boolean
+switchOn(ObjectTh):void
+switchOff(ObjectTh):void
+isSwitchedOnFor(ObjectTh):boolean
...
Responsibilities:
- Represent an instance of an Aspect

Figure 4.5: AspectObject extends ObjectTh

Thorn provides us with a gradual (optional) static type system, in that we can annotate arg with the type we wish to restrict it to. If we only wanted the argument to be of type int, then we can alter the above pointcut as follows:

```
before: exec jp[(sense)] args(arg:int)
```

The aspecting language is fully compatible with the Thorn type system. It will not advice sense if the argument to sense is not of type int. The existing type checking mechanism that was in the interpreter was reused for this.

4.2 OOP / AOP unified aspecting

As discussed in section 3.2, there are advantage to following an instantiation model of aspects which closely resembles how objects are instantiated.

4.2.1 Aspect instantiation

An AspectObject (the runtime representation of an instance of an aspect) is an extension of an ObjectTh. As such, it also has e.g. a ClassStatic, that has the fields / methods of that class / aspect.

An AspectObject has the additional feature of tracking the Thorn objects it is switched on for, as well as inserting itself into the cache of turned on objects for ObjectTh instances.

4.2.2 *azpct* keyword

During the execution of a method, the *this* keyword in Thorn has the obvious meaning it does in other languages, which is that is a reference to the object whose method is being called.

In Java, this is an optional keyword to access fields and methods of the *this* object and is only required when there is a ambiguity with an identifier, i.e. an argument and an object field have the same name, so the *this* keyword is used to access the object field. This is in contrast to Thorn, which requires the *this* keyword to always be used when accessing fields and methods of the *this* object.

In AOD-Thorn, we also have the *azpct* keyword, which refers to the aspect object in whose context the current advice is executing. More on this later.

4.2.3 *this and origin* keywords in the context of Aspects

In AOD-Thorn, when executing advice, the this keyword will refer to the target object of the joinpoint. In other words, if we have the following piece of code:

class C1() val tgt = C2(); def foo() = tgt.bar();

... before: exec jp[C2.bar] args() //advice

Then, while the advice is executing, the *this* keyword references the instance of C2 named tgt, and the *origin* keyword refers to the instance of C1 which made the method call on tgt. In a sense, the *this, origin* and *azpct* objects become collaborators.

4.2.4 *origin* keyword

As described before, the origin keyword in advice is used to reference the object where a method invocation / field access came from.

The original Thorn does not have a concept of an origin, and so, the interpreter had to be modified to include the origin in the stack frames when executing advice.

This required that the operational semantics of field accesses and method invocations are modified, so that the *origin* is included in the stack frame.

For example, the original routine for method invocation had the following signature:

Listing 4.2: fisher.runtime.ObjectTh

```
...
private Thing invokeMethodInternal(Thing[] args,
  Syntax src, MethodSig sig, FunBody funbody)
```

while the modified routine has the following signature:

Listing 4.3: fisher.runtime.ObjectTh

```
...
private Thing invokeMethodInternal(Thing[] args,
  Syntax src, MethodSig sig, FunBody funbody,
  Thing origin)
// the origin is the "Thing" where the method call came from
// "All runtime Thorn objects are Things", i.e. they all derive
// from the fisher.runtime.Thing class.
```

The routines for field accesses were dealt with in a similar manner. These changes needed to be propagated throughout the code base wherever method calls and field accesses were handled.

The operational semantics of method calls and field accesses were changed, even when not giving advice, as there needed to be a way for the origin to be included in the stack frame on advice invocation.

4.2.5 BIFs: AddToAsp(AspectObject, ObjectTh), DelFromAsp(AspectObject,ObjectTh)

The addToAsp BIF will take a selective AspectObject and an ObjectTh. It will add the ObjectTh to the list of objects that the aspect instance should advise (this list is located in AspectObject), and it will add the AspectObject to the list of turned on aspect instances that are advising ObjectTh (this list is located in the ObjectTh).

For example, to instantiate an aspect Asp and then add an object to it, we can do the following:

```
aspect selective Asp{...}
class C{}
...
//instantiate
//the aspect Asp
asp = Asp();

//instantiate the object
//to be advised
obj = C()
```

```

addToAsp(obj, asp);
```

To make an aspect no longer advise an object, the delFromAsp does the opposite of addToAsp. It can be used as follows:

```
delFromAsp(obj, asp);
// aspect instance asp
// no longer advises object
// obj
```

4.2.6 BIFs: TurnOnAsp(AspectObject), TurnOffAsp(AspectObject)

The *turnOnAsp* and *turnOffAsp* BIFs work with unselective aspect instances. As described in more detail later, there is a global table mapping from unselective aspects to a list of their instances. *turnOnAsp* and *turnOffAsp* add and remove the aspect instance to this table. Note that when an unselective aspect is constructed, it is automatically added to this table.

There may be a need for a *turnOnAsp* and *turnOffAsp* BIFs which are used for selective aspect instances. Turning off a selective aspect instance would mean that it can be turned on later, without it losing the list of objects that it contained. This can be done easily (i.e. just check that the aspect instance is turned on before executing its advice). It wasn't implemented to keep the semantics of advice execution simpler, as well as it not being a particularly interesting feature, although if it was requested by users, the feature could be added with a few lines of code.

4.2.7 Advice execution

Earlier (section 2.1.6), we described the weaving process which matches the joinpoint identifier to several lists of advice AST trees, and we iterate through those lists to evaluate the AST trees.

However, we did not talk much about how exactly the advice is executed.

4.2.7.1 Selecting the azpct object

When a joinpoint is being executed, we find a list of advice ASTs, and evaluate the ASTs with the context of the modified stack frame described previously. Each advice AST (AdviceDecl) is a child of (and references) an AspectDecl.

AOD-Thorn allows us to instantiate both selective and unselective aspect instances, both of which are dealt with differently.

Unselective aspect instances When an unselective aspect is instantiated, it registers itself into a table δ which maps from the fully qualified aspect name to a list of unselective aspect instances for the described aspect. δ applies to all the classes which contain the joinpoints defined in the pointcuts in the aspect, so it does not make sense to store it with a specific Thorn object or class, and is therefore a global table.

During the execution of a joinpoint, for each AdviceDecl that applies to the joinpoint, if it is discovered that the AspectDecl the AdviceDecl belongs to is unselective, then the δ is consulted for a list of instances of that aspect (which includes all the instances of the aspect). Then, for each aspect instance *ai*, the advice is executed with the *azpct* set to reference *ai.*

Selective aspect instances When advice belonging to a selective aspect is being executed, the advice execution loop will only execute advice in the context of a subset of the instances of the aspect.

When an object is added to a selective aspect instance, the object caches the the aspect instances which are switched on for it. Each object holds a table mapping from an AspectDecl to a list of aspect instances (*List<AspectObject>*) for that AspectDecl.

Then, for each AspectObject in the list, the AdviceDecl is executed with the AspectObject being pointed to by the azpct field of the stack frame.

4.2.7.2 Method / advice AST evaluation

When we execute a method in Thorn, a stack frame[8] is used to hold the values of arguments as well as the *this* reference, and the method AST is evaluated[9] in the context of this stack frame (and, of course, its parents).

An advice AST is just like a method AST, however, it has the additional features that it can refer to the *origin* and *azpct* objects, using the respective keywords. To implement this, one of the steps was to modify the stack frames in the context of which advice execute in, so they include a reference to the *origin* and *azpct* objects.

Thus, the semantics of advice execution and method execution are different, since method execution is oblivious to the origin and azpct objects, they are only useful in the context of advice execution.

Lets look at this in more detail by looking at the operational semantics. The reader may find the notation overwhelming if they are not familiar with work similar to L1/L2[15] by Prof. Drossopoulou (the notation has been altered to the authors taste), just below the semantics is a line by line explanation in English, which the reader may find useful to read first.

[8]fisher.eval.Frame
[9]by fisher.eval.Evaller

A key to symbols the reader may find useful (but reading about L1/L2 may still be necessary):

χ	$=$	heap
θ	$=$	stack
e	$=$	expression
m	$=$	method
ι	$=$	an address (can be used to lookup an object in the heap)
ϕ	$=$	origin
ξ	$=$	joinpoint information

Some parts of the semantics which do not relate to the aspecting have been omitted. For example, the *this* pointer in the stack is usually set to the object evaluated by e_0. This has been omitted as it is not relevant to aspects. The focus of the following is the changes made to the semantics, rather than the absolute semantics, in order to simplify an explanation of the changes that were made to AOD-Thorn. We do not attempt to fully describe the semantics of Thorn itself.

$$\begin{array}{ll}
1: & e_0, \theta, \chi \rightsquigarrow \iota, \chi^1 \\
2: & e_1, \theta, \chi^1 \rightsquigarrow \iota^2, \chi^2 \\
3: & \mathcal{M}(c, m) = m(arg)\ Adv_{before}\ Adv_{after}\ adv_{around}\{e\} \\
4: & doAdvices(Adv_{before}, \theta[\phi = \iota_\phi, \xi = m], \chi^2) \rightsquigarrow \chi^3 \\
5: & adv_{around} != null \\
6: & doAroundAdvice(adv_{around}, \theta[e_{orig} = e, \phi = \iota_\phi, \xi = m], \chi^3) \rightsquigarrow \iota^3, \chi^4 \\
8: & doAdvices(Adv_{after}, \theta[\phi = \iota_\phi, \xi = m], \chi^4) \rightsquigarrow \chi^5 \\
0: & \dfrac{}{e_0.m(e_1), \iota_\phi, \theta, \chi \rightsquigarrow \iota^3, \chi^5}\ (meth,\ with\, around)
\end{array}$$

$$\begin{array}{ll}
1: & e_0, \theta, \chi \rightsquigarrow \iota, \chi^1 \\
2: & e_1, \theta, \chi^1 \rightsquigarrow \iota^2, \chi^2 \\
3: & \mathcal{M}(c, m) = m(arg)\ Adv_{before}\ Adv_{after}\ adv_{around}\{e\} \\
4: & doAdvices(Adv_{before}, \theta[\phi = \iota_\phi, \xi = m], \chi^2) \rightsquigarrow \chi^3 \\
5: & adv_{around} = null \\
7: & e, \theta, \chi^3 \rightsquigarrow \iota^3, \chi^4 \\
8: & doAdvices(Adv_{after}, \theta[\phi = \iota_\phi, \xi = m], \chi^4) \rightsquigarrow \chi^5 \\
0: & \dfrac{}{e_0.m(e_1), \iota_\phi, \theta, \chi \rightsquigarrow \iota^3, \chi^5}\ (meth,\ no\, around) \quad .
\end{array}$$

The English description of the above:

0: We wish to make a method call $m()$ on an object to which we can gain a reference to by evaluating the expression e_0, with an argument to which we gain a reference by evaluating the expression e_1. θ represents a stack frame, which (abstractly) is a table mapping from identifiers (such as *this, azpct,* argument1, etc..) to their values. χ represents the heap, which (abstractly) maps from addresses (represented by ι) to objects[10]. ι_ϕ refers to the object where this method call is taking place from (the origin), i.e. the object which contains the expression $e_0.m(e_1)$. This method call evaluates (represented by $\rightsquigarrow$) to an address ι^3, and the heap is modified to χ^5 during this whole operation.

1: Evaluate e_0 in the context of stack frame θ and the initial heap χ. This evaluates to address ι. and χ is transformed to χ^1.

2: Similar to line 1

3: The function $\mathcal{M}(c, m)$ with return the MethDecl associated with the class of the object $\chi^2(\iota_2)$.[11] The MethDecl contains a set of before advices Adv_{before} and a set of after advices Adv_{after}, as well as a single around advice Adv_{around} (all of these may be null). The original method body is represented by $\{e\}$. There can be a set of before and after advices, because neither of these are allowed to modify the control flow of the method being advised, i.e. they cannot return from it. However, around advice is not only able to advise the method to return, it is also allowed to advise it to proceed()[12], so the following type of code is possible:

```
around: exec jp[meth] args(arg){
  timeBefore = timeNow();
  ret = proceed();
  timeAfter = timeNow();
  timeTaken = timeAfter - timeBefore;
  log("Method meth took " + timeTaken +
      " and returned value: " + ret);
  }
```

4: *doAdvices*(...) is more fully explained later. Simply evaluates all the advices in the Adv_{before} set. The stack frame used to evaluate the advice has the ϕ identifier (which identifies the *origin*) set to the address of the origin. Further, joinpoint information is set (i.e. the string version of the seal) in ξ, in this case the name of the method.

[10]i.e. $\chi(\iota)$ evaluates to an object at address ι

[11]This type of lookup doesn't actually happen in the implementation, there is no need to look anything up as its provided during the method invocation in the interpreter, but the function is used here as an abstraction over this, and for convenience

[12]obviously only one advice should be able to call proceed() or return, since this can only be done once

5: Check if there is any around advice.

6: Evaluate the around advice, with the e_{orig} identifier pointing to the original method body, i.e. what would be executed if the around advice calls proceed(). The function $doAroundAdvice(...)$ is explained more fully later.

7: There is no around advice, so execute the original method body.

8: Evaluate the after advices.

The above set of semantics used functions $doAdvices(...)$ and $doAroundAdvice(...)$. They are defined as follows:

```
1 : $\mathbb{T}(adv, \iota) = \{a | a \in \iota[turnedOn(adv.parent)]\} \cup UnselObj(adv.parent)$

2 : $doAdvices(Advices, \theta, \chi^0) = \{$
3 : $|Advices| = k$
    $\forall i \in 0..k-1\{$
4 : $\mathbb{T}(Advices[i], \theta[\Theta]) = AO$
5 : $|AO| = l$
        $\forall j \in 0..l-1\{$
6 :     $Advices[i], \theta[\alpha = AO[j]], \chi^{i,j} \rightsquigarrow \chi^{i,j+1}$
    $\}\}\}$


7 : $doAroundAdvice(adv, \theta, \chi^0) = \{$
8 : $\mathbb{T}(adv, \theta[\Theta]) = AO$
9 : $|AO| = k$
        $\forall i \in 0..k-1\{$
10 :    $adv, \theta[\alpha = AO[i]], \chi^i \rightsquigarrow \iota, \chi^{i+1}$
    $\}\}$
```

1: Objects contain a table that maps from the AdviceDecl, to a list of instances of the aspect which contain this AdviceDecl (as in figure 4.5). This function will return all the turned on aspect instances (both selective and unselective) for the object referenced by ι. The aspect instances are of the type which is the parent (AspectDecl) of AdviceDecl adv. Furthermore, there is a global table $UnselObj$ which maps from an unselective AspectDecl (which is the parent of the AdviceDecl adv) to turned on instances of this unselective aspect.

2: *doAdvices*(...) takes a set of advices, as well as the stack and heap, and modifies the heap.

3: There are k advices.

4: For each advice, get the turned on aspect instances of the *this* object (Θ is the *this* pointer) that contain the advice.

5: There are l turned on aspect instances to be used

6: Evaluate an advice, after setting the *azpct* (α) identifier to point to the aspect instance

7: Same as before

8: Same as before

9: Same as before

10: Execute the around advice. Notice that a ι is returned on each invocation, so depending on how many turned on aspect objects are, the original method body may be executed several times. If this proves problematic, it is easy enough to warn the programmer that several aspect instances are giving around advice to an object.

4.2.7.3 AspectObject tables, more formally

In the above semantics, the function $\mathbb{T}$ depends on the table *turnedOn* and *UnselObj*. Below, we describe how these tables are generated.

We give the semantics involving these tables, while ignoring stacks (they're not needed in the below rules).
It assumes the *turnedOn* table (recall *inst(x)* is the set of instances of x): $turnedOn : AspectDecl \to \mathcal{P}(inst(AspectDecl))$. Also the *UnselObj table*: $UnselObj : AspectDecl \to \mathcal{P}(inst(AspectDecl))$.

$$\frac{\iota'_{selective}[aspDecl] = asp}{addToAsp(\iota, \iota'_{selective}), \chi, UnselObj \rightsquigarrow \chi'(\iota)[turnedOn(asp) \cup \iota'_{selective}], UnselObj}(addToAsp) \tag{4.1}$$

$$\frac{\iota'_{selective}[aspDecl] = asp}{delFromAsp(\iota, \iota'_{selective}), \chi, UnselObj \rightsquigarrow \chi'(\iota)[turnedOn(asp) \backslash \iota'_{selective}], UnselObj}(delFromAsp) \tag{4.2}$$

$$\frac{\iota'_{unselective}[aspDecl] = asp}{turnOff(\iota'_{unselective}), \chi, UnselObj \rightsquigarrow \chi, UnselObj(asp) \backslash \iota'_{unselective}}(turnOff) \tag{4.3}$$

$$\frac{\iota'_{unselective}[aspDecl] = asp}{turnOn(\iota'_{unselective}), \chi, UnselObj \rightsquigarrow \chi, UnselObj(asp) \cup \iota'_{unselective}}(turnOn) \tag{4.4}$$

4.1 When we use the addToAsp BIF, it takes an ObjectTh ι and an selective AspectObject $\iota'_{selective}$. The *turnedOn* table is modified by adding the selective AspectObject to the list of turned on AspectObjects which are instances of the aspect described by asp.

4.2 Similar to addToAsp, except now the AspectObject is removed rather than added.

4.3 The unselective AspectObject $\iota'_{unselective}$ is removed from the turned on unselective aspect instances table in *UnselObj*.

4.4 Same as for turnOff, except now $\iota'_{unselective}$ is added to the *UnselObj* table. It should be noted that when an unselective aspect is being constructed, i.e. through syntax *MyUnselAspect*(), it will add itself to the *UnselObj* table.

joinpoint

4.2.7.4 Semantics for the this, origin, azpct, joinpoint() and proceed() syntax

$$\frac{}{this, \iota_\phi, \theta, \chi \rightsquigarrow \theta[\Theta], \chi}(this) \qquad \frac{}{origin, \iota_\phi, \theta, \chi \rightsquigarrow \theta[\phi], \chi}(origin)$$

$$\frac{}{azpct, \iota_\phi, \theta, \chi \rightsquigarrow \theta[\alpha], \chi}(azpct) \qquad \frac{}{joinpoint, \iota_\phi, \theta, \chi \rightsquigarrow \theta[\xi], \chi}(joinpoint)$$

$$\frac{\theta[e_{orig}], \theta, \chi \rightsquigarrow \iota, \chi^1}{proceed(), \iota_\phi, \theta, \chi \rightsquigarrow \iota, \chi^1}(proceed)$$

4.2.7.5 Field access AST evaluation

Unlike for transparently distributed aspecting, field access interception for local aspecting was fully implemented, in fact in a very similar way to how methods interception works, and follows very similar semantics, so they have not been written out.

Essentially, the same procedure is followed, i.e. run before advice before the field access happens, then run the around advice if it exists, then run the after advice, while taking into account the obvious differences between method calls and field accesses. The *joinpoint*, *azpct*, *proceed*, *azpct*, *origin* and *this* constructs work as in method calls.

4.3 Transparently distributed aspecting

Transparently distributed aspects could have been implemented using several techniques given the code base as is.

Abstractly, the slave intercepts events (joinpoints) in the execution of the program which match the predicate defined in a pointcut. The slave informs the master of the event, and gives it necessary information (such as values of arguments to a method call) that the master needs to give the event advice. The master gives advice to the event, using some mechanism, either locally or at the slave machine.

This feature was highly experimental, and turned out to have some severe limitations (see section 5.7.1). However, in a restricted number of cases, it may be useful, and and with some possible (admittedly large scale) changes to the interpreter, it may even have useable performance.

4.3.1 RMI

Java RMI (Remote method invocation) allows one to write distributed applications where an application can invoke the methods of objects on a separate VM[58].

Instances of a class θ which implements an interface β that extends *java.rmi.Remote* can be registered on an RMI registry using a name assigned by the programmer.

Only those methods of β which are marked as throwing *java.rmi.RemoteException* can be invoked remotely. θ can have public methods other than those defined in β , but they cannot be invoked remotely.

Since transparently distributed aspecting is essentially a server / client distributed architecture consisting of a server machine with the master aspect running (on whom we can do the remote method invocations) and several clients (the slave machines), RMI was seen by the author as a simple mechanism which can be used to implement this functionality.

It is essentially an abstraction on taking certain actions in the server according to events described in messages being sent over the network.

4.3.2 Setting up a master aspect

After an aspect is instantiated, we can set it as the master aspect as follows:

aspectInst = Aspect(); masterAspect(aspectInst, "nameForAspectInst");

The masterAspect BIF is what we are interested in. The listing above demonstrates the arguments required by the masterAspect:

1. The aspect instance that is to be made the master
2. A name chosen by the programmer for the aspect master. This name will be used by slave aspects that wish to form a link with the master.

This basically takes the AspectObject aspectInst and it will create a *stub* from the AspectObject, which is then exported onto the RMI registry, under the name provided as the second argument. Methods of AspectObject such as invokeMethod (as the name suggests, this is required to invoke methods on the AspectObject) are included in the stub, which involved a lot of refactoring.

4.3.2.1 RMI stubs

An RMI *stub* is essentially a proxy for an object. [58, 3]. The client can obtain the stub by contacting the RMI registry, and asking by name for the stub required.

The client can then invoke methods on the stub, which shares the β interface with the class that it is the proxy for. When the client invokes methods on β which can throw *java.rmi.RemoteException,* then the RMI subsystem will take the necessary steps for the method to be invocated on the server.

Of course this requires an RMI registry to be started at the machine with the master aspect. This is done lazily, an RMI registry is started up the first time that the *masterAspect* BIF is invoked in the application being executed by the interpreter.

4.3.3 Setting up a slave aspect

A slave aspect instance is setup as follows:

slaveAsp = slaveAspect("hostWithMastersRegistry", "nameForAspectInst");

// if the aspect is selective, we can add // objects to it as usual: addToAsp(object, slaveAsp);

As we can see from the above snippet, we simply need to invoke the slaveAspect method, which takes two arguments:

1. The address where the RMI registry holding the stub for the master aspect instance can be found, i.e. a “hostname:port” string
2. The name the master aspect instance is bound to in the registry

The function returns the proxy for the master aspect. If its an unselective aspect, then it is added to the unselective aspect table (the table as defined in section 4.2.7.1), otherwise it is added to the aspect object cache (see section 4.2.7.1) of Thorn objects using the addToAsp BIF.

Once this is done, advice is given as usual, i.e. when a joinpoint such as a method execution is hit, it checks whether it matches any pointcuts and if it does then it executes advice in the context of a frame which has an *azpct* reference inside it. However, instead of the *azpct* pointing to a normal AspectObject, it contains a stub for an AspectObject living on the master machine.

4.3.3.1 Slave objects

When executing advice, objects other than those referenced by *azpct* or *origin* or *this* can be used (e.g. one of the fields of the *origin* or *this* object, or the arguments passed into the method being advised). This proves problematic when executing advice, since the master aspect instance may not have access to these objects, if they lie on the slave machine.

For example, lets say the advice contained the following snippet:

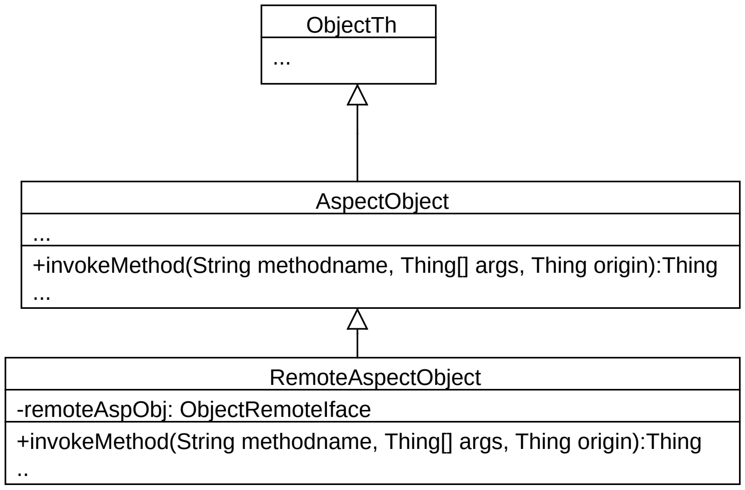

ObjectTh
...
AspectObject
...
+invokeMethod(String methodname, Thing[] args, Thing origin):Thing
...
RemoteAspectObject
-remoteAspObj: ObjectRemoteIface
+invokeMethod(String methodname, Thing[] args, Thing origin):Thing
..

Figure 4.6: The relationship between Thing, AspectObject, RemoteAspectObject

before: exec jp[(sense)] args(arg) azpct.log(this.name + " sensed: " + arg);

Clearly, the master instance requires access to the *this* object as well as the *arg* object. These objects are turned into RMI stubs, and are sent to the master aspect, which can invoke remote operations on them. The operations are evaluated on the slave machine where the objects lie, and the result is returned to the master. In the above example, the master would invoke remote operations to read the string representation of *this.name* and *arg.* The string representation procedure is ran on the slave machine, and the result of this procedure is returned to the master.

This can be described as a distributed heap, with the distributed heap consisting of the individual heaps of applications where the master and slave aspect instances live. We were vague about how exactly e.g. method invocation works. A more exact version follows.

4.3.4 Method invocation

When invoking a method on a ObjectTh or AspectObject that live on the master, we need to make all the referenceable objects from both sides remotable.

For example, if advice on the slave is as follows:

```
before: exec jp[joinpoint] args(arg){
        ...
        ret = azpct.meth(this.field);
```

The *azpct* is a RemoteAspectObject. In the implementation, it simply subclasses an AspectObject, it acts as a proxy for the AspectObject that is on the master machine. When we invoke a method on the *azpct* as above, this method invocation is passed to the AspectObject that it is the proxy for (i.e. as determined by the slave BIF described earlier).

However notice that we aren't just dealing with the *azpct* RemoteAspectObject here, we are also dealing with arguments and the returned value from the method.

We can't just serialise the arguments / returned object and send them over the wire, the object may be part of a huge object graph, this would be a severe limitation on the performance. Furthermore, as this is a distributed application, the master may wish to invoke operations on objects on the slave machine.

For example, we may have advice of the form:

```

 // the passed in object has method called
 // getName
 def logCalled(obj) = {println(obj.getName());}

 before: exec jp[joinpoint] args{arg}{
     ...
     azpct.logCalled(this);
```

It should be apparent that the master aspect instance needs to be able to call the method getName() on the object passed into the logCalled method. However, that object was never exported to the RMI registry, nor is it practical to export every object to the RMI registry, when retrieving it, one needs to know the name used to export the object.

The workaround needed a lot of digging into the RMI documentation. It turns out that RMI stubs can be used anonymously, i.e. without needing to export them to an RMI registry and name them explicitly. A stub contains the IP address of the machine it was born on, as well as the port of a service which maps from the stubs to the stubbed object.

We realised that stubs should not be too big, they are not part of large object graphs. As such, it is easily possible to serialise a stub and send it from machine A machine B. When machine B invokes operations on the stub, the stub can deliver these operations back to machine A (the stub knows its way home).

The Thorn remoting mechanism was designed with both pass-by-reference (through stubs) and pass-by-value (serialise object graph) semantics.

At the interpreter level of communication (the modifications made to the interpreter), we are using Java RMI, which uses pass-by-value semantics, and pass-by-reference is emulated by serialising stubs.

When one interpreter invokes an operation on the running instance of another interpreter (i.e. invoking the method invokeMethod(...)), we need to deal with arguments as well as return values from the remotely invoked method. This means that that live on interpreter A need to be transferred to interpreter B.

The relevant part of the signature of the method RemoteAspectObject.invokeMethod(..) is *Thing invokeMethod(String methodName, Thing[] args, ..).* The returned *Thing* is the Thorn object returned by the Thorn method described by *methodName* (the method arity and argument types are worked out using the *args*[] array), the *args[]* array is the array containing the Thorn objects which are used as arguments for the Thorn method described by *methodName.*

The *pure* syntax in Thorn can be applied to classes (i.e. *class C :pure{})* which are immutable, and whose fields only refer to *pure* objects. All immutable built in types (such as the primitive objects, i.e. StringTh, IntTh etc) are *pure.* Furthermore, *pure* objects are not allowed to perform operations which are marked as impure, such as IO (e.g. the BIF *println()*) is marked impure.

We chose to use pass-by-value semantics for *Things* which are marked as *pure,* and we chose to use pass-by-reference semantics for *Things* which are not *pure.*

The *pure* syntax is added to classes by Thorn programmers when they wish to make it possible to send objects of that class between *components.* These objects are immutable, and furthermore they cannot cause IO side effects.

This gives us a strategy in which we can pass stubs for *Things* which are not pure and pass the serialised version of *Things* which are pure.

So for example, if we have a class as follows:

```
class C(){
  def printX() = {println("X");}
  def getStr() = {return "str"}
}
```

This class is not declared as pure, and it can't be, it uses *println().* If an instance of class C was either the argument or the return value for a method, it would be sent over pass-by-value semantics. However, if one invokes the *getStr()* method on the stub of a class C object, the remote machine will serialise the StringTh object (which is pure) and return it using pass-by-value semantics.

This means that all side effect causing operations are done where the object graph / IO stream where the non pure object lives.

The mechanism is useful for programmers who want to convert their single threaded programs into concurrent / distributed programs. They can gradually add the *pure* syntax to objects which need to be eventually passed between the JVM instances using e.g. components.

The strategy means that objects on different machines can refer to each other as if they were local, with no special syntax needed to do so.

However, the weakness of this strategy arises when we pass the Thorn - Java barrier, i.e. use *Javaly* (described in section *2.3.1.7*). Javaly does not account for Java objects which are mutable / cause IO (i.e. impure). Having said this, Thorn components also suffer from this problem.

4.3.5 Field access

Object field accesses are currently implemented in a way that is not too friendly to them being invoked remotely. For example, they require stack frames (even though they don't use them) and the syntax tree where the field access came from (used only when throwing exceptions). Furthermore, they require full seals of the field that is to be accessed, even though the field access routines only use the string values of the seal.

This, coupled with the somewhat complex inheritance hierarchy that these field accesses are part of (large inheritance hierarchies seem to be very common in the Thorn interpreter code), means that relatively heavy refactoring would be required to make field accesses remotely invokeable.

The work with method invocations identified some key limitations of this approach when it is used in Thorn, as discussed in the evaluation chapter.

Because of the limitations and the amount of relatively uninteresting work required to make field accesses work, field accesses were not implemented. Instead, one can simply wrap the field access in a method. For example, while the following code in advice to a slave won't work:

```
var name;

pointcut{
  println(azpct.name)
}
```

the following will:

```
var name;
def getName()= return name;

pointcut{
  println(azpct.getName());
}
```

After doing the necessary refactoring, it would be easy to implement field accesses as required.

Chapter 5

Evaluation

In this section, we attempt to evaluate the work that was done as part of this project.

5.1 Relative to other AOP suites

We list below the functionality common to mature AOP implementations, and how AOD-Thorn stacks up against them. It should be noted that there is no competing AOP solution for Thorn, so it was required that

5.1.1 Aspect locations

As discussed before, there are a a number of advice locations which are common to most AOP suites[14, 5], namely:

1. **Before** - run advice before joinpoint is executed, but cannot alter whether joinpoint runs or not. Fully implemented in AOD-Thorn
2. **After** - run advice after joinpoint is reached, cannot alter whether joinpoint runs or not or what the joinpoint returns. Fully implemented in AOD-Thorn.
3. **Around** - start to run before the joinpoint. Use a *proceed()* type statement which will proceed the joinpoint, and return the value returned by the joinpoint (if its a method call). May prevent the joinpoint from proceeding, and can also modify the value returned by the joinpoint. All of this is fully implemented in AOD-Thorn.

4. **After returning** - advice runs if the joinpoint (a method execution) returns a value described in the pointcut. This is syntactic sugar for an around advice with the return value tested to determine if the advice should be ran. This syntactic sugar was not implemented as it is not especially interesting.

5. **After throwing** - advice runs if the joinpoint throws an exception, described in the pointcut. This functionality was not implemented, as Thorn as of yet does not have a satisfactory exception throwing mechanism: there is no hierarchy of exceptions, all exceptions are of the same type, except the string string representation is different. It is the authors opinion that if Thorn was not scrapped, this mechanism would have changed drastically and therefore there was no point in implementing this functionality.

Given the above list, AOD-Thorn supports all the common advice locations, with the exception of after throwing, which is due to the immaturity of Thorn. If Thorn had a more mature exception throwing mechanism, it would surely not be much more difficult to implement the advice location implementation for it than for the other cases.

5.1.2 Joinpoint types

There are 5 kinds of (very useful to intercept) join points common to mature AOP suites[5]. We list them below and describe how they're supported in AOD-Thorn:

1. **Method execution -** intercept the execution of a method, where the *this* object in the stack frame used when executing advice is set to the object which is the target of the method call. This is fully implemented in AOD-Thorn.

2. **Method call** - intercept the call of a method, i.e. the *this* reference in the advice refers to the object which has made the call. While specific syntactic sugar is not used for this functionality, the use of method execution interception paired with the *origin* syntax gives equivalent results. This is especially true due to the lack of information hiding mechanism in Thorn, i.e. advice to the target can call any methods on the origin or access any fields, as there is no way to make them private.

3. **Field set** - intercept the binding of a new value to a field. Fully implemented in AOD-Thorn, including the ability to report on the value which will be bound to it.

4. **Field get** - intercept the reading of a field. Fully implemented in AOD-Thorn, including ability to report on the value of the field.

5. **Exception handling -** basically catch clauses, points in code which handle exceptions. Not implemented in AOD-Thorn, due to the reasons given before. One suggest a large reduction of exception handling code, more than 90% reduction in catch statements in the applications studied[29]. The additions of proper exception handling to Thorn may benefit in the same kind as Java.

With the exception of exception handling (due to limitation of Thorn), AOD-Thorn directly handles or emulates the functionality of all the other joinpoint types.

5.1.3 Other important pointcut primitives

There are two pointcut primitives not discussed above, namely scoping and cflow/cflowbelow. A pointcut is a set of predicates which describe when an advice should be given to a joinpoint. There is need to be able to restrict joinpoints to some scope, e.g. joinpoint must be in an object of a given class / package, and the cflow/cflowbelow analyses the stack trace in order to work out whether the joinpoint should be matched.

Sidenote:usefulness of cflow/cflowbelow The cflow / cflowbelow pointcut primitives look at the route the program took to arrive to the current joinpoint. For example, it may only wish to intercept the pointcut if it occurred as a result of some method call down the stack trace.

For example, these primitives are often used to enforce that a field is only set during the initialisation of the program. "For example, the cflowbelow pointcut can be used to enforce that the field set is only during the init() procedure"[5]. This is often a problem experienced in Java when dealing with dependency injection. Classes with many fields that are to be injected are either forced to have long constructors (and sometimes several of them), or to expose a public setter. Clearly this is not an ideal solution, as it violates information hiding.

The cflow.. primitives can be used to ensure that the exposed field setters are only called as a result of being set by the DI solution, by looking at the stack trace. This solves the dilemma faced by the programmer of whether he uses bad design with very long constructors or violates information hiding principles. Further, there is no need for the programmer to scatter some kind of mechanism (possibly checking the stack, i.e. encapsulating code that checks Thread.currentThread().getStackTrace()) to check where the call is coming from.

Implementation of scoping and cflow/cflowbelow In AOD-Thorn, scoping was implemented through the use of regular expression matching on the seal of the joinpoint.

Due to lack of stack trace information in the Thorn interpreter, cflow / cflowbelow functionality was not implemented. It is however useful to note that one can simulate some of the benefits of these primitives. For example, to enforce that a field is only set once (which is close to the requirement that it is only set at initialisation / by DI solution), an aspect instance attached to the object can intercept the field sets, and use a boolean field to check whether the intercepted field has been set before or not. An aspect library can be written around this idea, making the task very easy.

However, cflow / cflowbelow seems to be associated with an AOP "advanced" feature, and are generally very expensive to evaluate, "even compared to other dynamic pointcuts"[48] and therefore probably not used as widely as the others, although we have no data to back up this claim.

5.1.4 Instantiation models

Other than cflow/cflowbelow, as discussed before, most popular AOP suites have 3 other instantiation models (ways to instantiate an aspect). How they work has been described in section 3.2.1.3. It should be noted that all of those types of instantiations can be simulated using our instantiation model: with our model, the developer has complete control over when aspects are instantiated, as opposed to the AOP framework. We describe how similar behaviour can be accomplished using the instantiation model in AOD-Thorn:

1. singleton: instantiate a single unselective aspect instance
2. perthis: in the constructor of the class which needs the perthis model applied to it, instantiate the aspect and add self to the aspect instance. Alternatively, move this code into an aspect, i.e. there would exist an aspect which intercepts the constructor, and gives it advice that it should instantiate an aspect and add itself to it
3. pertarget: move this model into an aspect, i.e. when the pointcutted joinpoint is executing, check to see if the target has attached to it

5.1.5 The limitations compared to other AOP suites

The above paints a positive picture, in that most of the commonly found features of mature AOP suites are implemented. Unfortunately there are some limitations.

5.1.5.1 Annotations

Possibly the biggest let down in AOD-Thorn is the lack of possibility to joinpoint selection based on some annotation.

For example, in AspectJ, it is possible to annotate fields, and then include in the pointcut definition that we wish to match fields with the given annotation. An example application for this would be a persistence framework, which would persist every write to field that has been annotated (i.e. to implement fault tolerance).

For example, lets think about a bank application which has the requirement that it contains the ID of a customer along with the account balance. Further, it has the requirement that, for legal reasons, the balance must be accurate at all times, including in the case of the server crashing, so it is not allowed to persist the page views every once in a while.

If AOD-Thorn could support annotations (or some other kind of source level metadata), then we could have a simple implementation the above as follows (of course real solutions are going to be more complicated where money is involved, but we present a simplified solution):

Listing 5.1: BankBalance.th

```

class Balance(customerID)(
 import faulttolerance.@checkpointed;
 @checkpointed(
   name=customerID + " balance")
 var balance;

 ....
)
```

Listing 5.2: CheckpointingAspect.th

```

module FaultToleranceAspects{
 import faultTolerance.@checkpointed;

 aspect unselective CheckpointingAspect{

    // before any field with
    // the checkpointed annotation
    // is set
    before: set jp[*] withAnnotation(@checkpointed ch) args(arg){
      // write the newly set field to the disk
      // given some checkpoint name and the value
      // the field will be set to, persist the new value
      // to the disk
      persistToDisk(ch.name, arg);
    }

    ...
}
```

Its easy to imagine a continuation of the above aspect, i.e. after a crash, when a customer object is being constructed using a list of customer IDs, the fields which are annotated as checkpointed will read the checkpointed value from the disk in some manner.

Why AOD-Thorn doesn't support this The reason is relatively simple, in that while the Thorn OOPSLA '09 paper promises an annotation system (as well as extensible syntax and so on)[6], those features are largely credited to the compiler rather than the interpreter. There doesn't appear to be a similar architecture present in the interpreter.

If the feature was added to the interpreter, the extension to AOD-Thorn to support this would not be too difficult to add.

5.1.6 Joinpoint information

Currently, when calling joinpoint() inside advice, a StringTh is returned (a Thorn String object), which contains the seal of the joinpoint, i.e. a method m in class C with arity 0 returns "C.m/0".

This leaves a lot to be desired compared to AspectJ / Java, which uses reflection that can get much more information about the joinpoint where the advice is running, i.e. when the equivalent of joinpoint() is called in AspectJ during the execution of a method, a *java.lang.reflect.Method* object is returned.

The implementation was left fairly extensible, advice execution operates in the context of a stack frame which contains a reference to a Thing (a Thorn object) which contains joinpoint information. While this is currently set to a StringTh object with the seal name of the joinpoint, this can easily be changed if Thorn had some sort of reflection API.

5.1.6.1 Ability to cross the Thorn - Java barrier

As a JVM language, it would be wise for Thorn to be able to call upon the extensive API present to Java. Indeed, the interpreter has such a mechanism, Javaly, as described in the background section.

Sometimes it is required to intercept calls to an API, i.e. JDBC as will be described in section 5.2.2. Currently this doesn't seem too big of a issue, calls to Java libraries need to be wrapped in Thorn objects.

However, we have seen from the experience of Scala[34], that it is possible for JVM languages to seamlessly interoperate with the Java API.

If such a seamless Java interoperation mechanism was added to Thorn, the aspecting extensions would be severely limited, and possibly shunned for their inability to go past this Thorn - Java barrier.

It may be possible write an aspecting mechanism which can pass this barrier. A language agnostic AOP suite has been researched as doable[28] for the .NET platform, so perhaps it may be required that a similar work is done for the JVM which can for example compile Thorn advice (if Thorn had a compiler) and give that advice to Java joinpoints.

5.2 Expressiveness

Expressiveness: "effectively conveying meaning or feeling" - from the Oxford Dictionary. We explore whether AOD-Thorn allows us to convey "meaning" or "feeling" of code more effectively.

We present some example code which benefit from aspecting in AOD-Thorn. It is relatively difficult to show the real power of AOP without a large application under consideration, after all, the problems of scattering and tangling really become a problem in complex applications rather than small ones. For this reason, the reader is asked to use their imagination and / or experience of previous software systems they may have come across. If they think about the problem at hand, and how it may become an application wide problem if it is replicated throughout the application many times, then the reader should start to really appreciate the elegance of AOP.

5.2.1 Person information example

We present the following class, which holds information about a person, and is thus a good example of a business object. The responsibility of the class is not only to hold information about the person (the core concern), but also to verify that the person has a legal name (i.e. make sure there are no numbers, space etc in the name). It should throw an error (which is assumed to be a function available in Thorn which generates a full stack trace) with the value of the argument if the argument could not be validated.

Listing 5.3: Person.th

```
class Person (ID){
  var forename;
  var middlename;
  var surname;
  var title;

```

```
    def setForename(forename){
      if(isValidName(forename)){
        log("Person.setForename(" + forename + ")");
        this.forename := forename;
      }else{error("Invalid input: " + forename);}
    }

    def setMiddlename(middlename){
    if(isValidName(middlename)){
        log("Person.setMiddlename(" + middlename + ")");
        this.middlename := middlename;
      }else{error("Invalid input: " + middlename);}
    }

    def setSurname(surname){
      if(isValidName(surname)){
        log("Person.setSurname(" + surname + ")");
        this.surname = surname;
      }else{error("Invalid input: " + surname);}
    }}



    def setTitle(title){
      if(isValidTitle){
        log("Person.setTitle(" + title + ")");
        this.title = title;
      }else{error("Invalid input: " + args);}
    }}



    def isValidName(name){
      return name.matches?("^[a-zA-Z]+$")
    }

    def isValidTitle(title){
      titles = ["Mr", "Ms", "Mrs", "Dr", "Prof", "Lord"]
      return name.matches?(
    }


    def log(msg){
      //Extremely simple println logging.
      //Realistic application would use a
      //logging library
```

```
            println(msg);
        }
    }
```

A solution in AOD-Thorn We take the cross cutting concern of logging, verification and throwing an error into an aspect that utilises joinpoint information to decide how to do verification. Then it logs the event, again using the joinpoint information.

Lets see the results:

Listing 5.4: Person.th

```
      class Person (ID){
      var forename;
      var middlename;
      var surname;
      var title;
 }
```

Listing 5.5: Aspects.th

```
module Aspects{
  aspect unselective LogAndVerify{

    around: set jp(Person.\.*name) args(arg){
      if(isValidName(arg)) log(joinpoint()+":"+arg);
      proceed();
    }

    around: set jp(Person.title) args(arg){
      if(isValidtitle(arg)) log(joinpoint()+":"+arg);
      proceed();
    }


    def isValidName(name){
      return name.matches?("^[a-zA-Z]+$")
    }

    def isValidTitle(title){
      titles = ["Mr", "Ms", "Mrs", "Dr", "Prof", "Lord"]
      return name.matches?(
    }


    def log(msg){
      //Extremely simple println logging.
      //Realistic application would use a
      //logging library
      println(msg);
    }
}}
```

What did we achieve?

1. Reduction in scattering - there is no longer an almost copy / paste version of the log statement everywhere
2. Reduction in tangling - there is no longer an unnatural link from the Person class to what would be a logger class in a real application
3. Reduction in line count - the total lines for the Thorn solution is 55, the total for AOD-Thorn solution is 36
4. No need for setters - setters were no longer needed, we now have field access interception.
5. No longer violate the "do one thing" and "single responsibility" principles[32] - the Person class now just holds data. It no longer holds data and the methods no longer verify input and logs input and throw errors when input is wrong.

All of the above make the code more modular / reusable. The class can now be used in other projects where:

1. The logging requirements are completely different - i.e. they might not need it at all or use a different logging library
2. They may have different input verification procedures, for examples names may be in Cyrillic, not Latin
3. The exception handling is completely different, it may not have been acceptable to catch the exceptions thrown by the Person class

5.2.2 Security example - SQL verification

Lets look at another example, which may well help in securing the application. SQL injection is a well known exploit used in web applications, where user input is not sanitised and malicious input may make its way into SQL queries made by the application. Furthermore, sometimes bad queries may

The usual solution to this is to sanitise inputs wherever they appear before they make it into an SQL query. For example (we assume the existence of a database connectivity library, the example is adapted from[62]):

Listing 5.6: queries.th

```
module queries{
 fun userDetails(user){
    if(isClean(user)){
     return "select * from users where uname="+user+";";
    }else{error("unclean input");}
 }
  ....
}
```

Listing 5.7: webform.th

```
class UserInfoForm{
  import sql.*;
  import queries.*;
  import dblayer.conndetails;

  def getUserinfo(user){
    query = userDetails(user);
    // execute the query given the
    // connection details and the query
    return SQLConn.exec(conndetails ,query);
  }
```

Now say that there are hundreds of queries, and at some point the developer forgets to sanitise the input into the query. For example, if we didn't sanitise the query for userDetails(), and somebody gives an input in an online form like: "'a'; drop table 'users';". This is clearly an SQL injection, all because a query was not sanitised.

This is a scattering problem. Sanitisation code must be scattered throughout the codebase, wherever SQL queries appear. How could we get around this? One common solution is to write a wrapper around the sql library. I.e. we could have SQLConnSanitised.exec(conndetails, query), which is a proxy for the SQLConn class. However, this places a large burden on the developer to write his own wrappers around existing libraries, and they can still make a mistake and call the original unwrapped library.

Aspects can help us out here. We can have in AOD Thorn an aspect like the following:

```
module SQLAspects{
  aspect unselective SQLSanity{
    around: exec(sql.SQLConn.exec) args(arg){
        if(isClean(arg)){proceed();}
        else{throw error("error("unclean input"));}
}}}
```

This aspect will sanitise every sql query that goes through the the SQLConn class, and the application is longer vulnerable to SQL injection attacks. This did not need sanitisation code scattered throughout the code, nor did it need a wrapper around the sql library, wrappers can be tedious to write.

The example should demonstrate how we can easily enforce a design principle across the application.

5.2.3 Expressiveness / conciseness of the previous 2 examples

The above two examples have demonstrated the changes to expressiveness and conciseness that an AOP suite can bring to a language.

Firstly, the code was made more concise. There is no longer a need to scatter what are essentially the same set of statements throughout the code, i.e. logging, input verification and so on. For example, if we had a logging aspect that logged all the setters in a package (or module, the Thorn equivalent of packages), and we added a new bean to the package, there would be no need to copy and paste the logging code into it, because the aspect would take care of this. This is a clear win for our AOP suite.

Secondly, the code has become more *cohesive,* a bean now only has the responsibility of storing some data (i.e. about a person), it is no longer burdened with the responsibility of logging access etc. Similarly, the code that takes user input now no longer needs to check input for SQL injection attacks. Cohesiveness is a key OO design principle so this is a big win.

However, in a way it is less expressive.

For example, Joe the new programmer joined the team that had written the code in the first example. Joe added new fields housename, streetname and postcode. When Joe writes a unit tests for his new class, he notices that when he invokes the setters for streetname and housename, he sees that something is logging the setter invocations, even though Joe didn't add logging code. Further, when the postcode setter is invoked, there is no logging going on. This confuses him until one of his colleagues helpfully points out that the system uses AOP, and shows him the logging aspect.

As discussed later, problems like this can be solved using proper tools.

5.2.4 Expressiveness / conciseness of instantiable / selective aspects

We draw attention back to the examples in section 3.2 and 3.3, namely the parts where the user instantiates aspects and where they use selective aspects as opposed to using unselective ones.

5.2.4.1 User instantiable aspects

Unlike most AOP suites, there is no asymmetry in AOD-Thorn which treats the instantiation of aspects and objects differently: aspects can instantiate objects and objects can instantiate aspects.

This lowers the conceptual burden on the programmer not familiar with AOP, they have always known how to instantiate an object. It is easy to express when an aspect should be instantiated, simply call its constructor. There is now no need to worry about the AOP suite instantiating too many aspects, i.e. if we used a perthis / pertarget.

5.2.4.2 Selective aspects

To tag an aspect as selective is more expressive than to go along with how other AOP suites deal with this issue. It appears to be a common problem for an aspect instance to only apply to a set of objects, however most AOP suites force the programmer to use a hand crafted solution using lookup tables.

The ability to make an aspect selective leads to code which is more concise as there is now no need for aspect membership tables and lookups. This can heavily reduce scattering across advices, as explained in section 3.2.3.

5.2.5 Expressiveness / conciseness of distributed aspects

As seen in section 3.3, exporting an aspect instance onto the RMI registry is very concise, a one line effort from the programmer who needs to add this to code ran on the master machine.

Similarly, the programmer needs to add one line to the programs running on the slave machines.

This is in contrast to an approach more idiomatic to the original Thorn, which would have required one to specifically write code for communication between the aspect instances at the master and slave machines.

Furthermore, it is worth noting here that Thorn communicates between components through pass-by-value semantics, whereas communication between intra-component objects is via pass-by-reference semantics. This clearly requires a programmer who has designed their program while in the mindset of a single machine architecture to do a lot of work to optimise the program for pass-by-value semantics when they move the program to a distributed architecture. After all, the objects that are being passed between the components could be part of large object graphs, serialisation and transfer of these graphs may take a prohibitively long time (and of course, this is made worse if the objects being transferred have mutable state, the state could be mutated on the remote machine, then the mutated object graph would have to be sent back so it could replace the original graph; references to objects on the original machine will need to be fixed, this could be non-trivial). There is further discussion about this problem later in the chapter.

Unfortunately what the language gains in conciseness, it loses in giving the programmer complete control over what happens.

For example, the programmer may want to exert control over the protocols used in the communication. They may also wish to have some syntax with which they can control where an aspect computation takes place (discussed later). While we started work on this, i.e. adding syntax into aspects that would allow the programmer to specify a certain method is to be carried out on the slave instead of the default master (i.e. slave.operation() vs azpct.operation()), this was later abandoned in the argument that this was starting to turn very much into a solution that was only marginally putting off the inevitable step of moving the communication mechanism to Thorn components.

5.3 Performance evaluation

When a new language feature is introduced into a language, usually the first concern is whether it makes the programmer truly more productive, but often the next concern is whether gains in programmer productivity are wiped out by losses in performance.

It is important that developers do not think the overhead of a feature is not too high, otherwise they may form a habit of not using it (i.e. the feature gets a stigma attached to it, making people take irrational decisions), even where the performance hit is insignificant compared to the rest of the application.

5.3.1 Interpreter performance

Before going further, it is important to remember just how slow the interpreter is compared to a bytecode compiler.

For example, to execute a method call in the Thorn script, the interpreter needs to make many, many method calls internally while it traverses the method call AST, creating a stack frame requires creating an object after several internal method calls, and so on...

The interpreter should not be thought of as a viable environment for applications with high performance requirements. However, the REPL environment and the fact there is no need to compile with the interpreter may be seen as a valuable productivity boost, so users may find it useful to develop with the interpreter where they check the correctness of their application, and then compile it when the application is ready for deployment.

As such, it is important that the semantics of language features on the compiler and the interpreter are equivalent, as users would not be happy to find that their application no longer works when it is compiled. If aspecting is added to the compiler, it is important that the aspecting behaves as it does in the interpreter.

During the development of AOD-Thorn, performance was not the highest priority, keeping the interpreter as simple as possible was seen as more of a priority, as it would enable an easier merge with the main interpreter if AOD-Thorn was ever decided to be adapted into the main stream interpreter. Further, it was apparent from the start that the theoretical complexity of any algorithms required for aspecting should not have large performance hit on an interpreter application once it was loaded. The AspectJ team strived to make the performance hit of aspecting versus a hand crafted solution to be virtually non existent, so it was apparent that aspects in general do not add much of a performance hit.

Nevertheless, some time was spent towards making the performance hit as small as possible, perhaps so the approach taken in the interpreter could be adapted without too many problems for the compiler.

Before benchmarking was done, it was the authors opinion that the performance hit should be extremely small due to the way aspecting was implemented (i.e. most objects relating to aspects are cached with the ObjectTh or the MethDecl, making them a field access away).

5.3.2 Microbenchmark results - local aspecting

Please note that all the times quoted from here onwards are in nanoseconds, using the Java System.nanoTime() timer. This uses the clock with the highest accuracy available on the CPU, and has a *resolution* (not to be confused with accuracy!) of 1 nanosecond. From previous experimentation on my computer, the clock is accurate to about 15 nanoseconds).

The benchmarks taken were inspired by [19, 47]. As field access has a very similar treatment to method calls, very similar results are expected.

The machine that the benchmark were ran on had the following specs:

Model Packard Bell EasyNote TM85

CPU Intel Core i3 M350 @ 2.27GHz

RAM 3GB

OS Windows 7 64-bit

Java version 1.6.0_24

5.3.2.1 Method call with no advice

The first microbenchmark that was ran was to check whether method calls that have no advice attached to them have any overhead added to them. The overhead would come from checking the AdviceDecl AST for before / after / around advice. There was a statistically insignificant difference in means of a benchmark run on the old Thorn interpreter and the AOD-Thorn interpreter.

Using a paired t-test, the benchmark determined a 9ns difference in mean execution time to a method call that took no advice, -3ns to 21ns at the 95th percentile.

It is important that the overhead for method calls with no advice is as low as possible, i.e. near to 0. If not, developers writing high performance are likely to reject the entire AOP suite, rather than just avoid giving advice.

However, it is also important to keep the language runtime implementation as simple as possible while the language is young and rapidly evolving. As discussed before, it is rather unlikely that developers of high performance languages would choose Thorn until it first matured in syntax, and then the interpreter / compiler also matured to a highly performant implementation. As such, the 9ns constant time added to a method call (see table 5.1) is seen as more than acceptable to the author, who further notes that some of this could be taken away at the expense of making the implementation of the interpreter more complicated

5.3.2.2 Method call - before advice

The second benchmark aimed to measure how much of an overhead is added by executing advice just before an advised method body is executed. It did this by making a second method call as the advice, and this was measured against the handcrafted version which made the second method call at the joinpoint.

This scenario can emulate an example where a logging statement logging that a method is called is moved into an aspect.

The results state a 105 to 134 nanosecond (see table 5.1) difference between the two ways of adding advice, which is a ~20% difference. Percentage wise, this is very favourable against e.g. early incarnations of AspectJ[19].

From the authors experience, it is unlikely that code which is in a tight loop will experience the problem of scattering or tangling, as its unlikely to be replicated throughout the code. Users would be advised to stay away from giving advice to method calls inside tight loops.

Its the authors opinion that for the vast majority of cases where advice is given to a method or a field access, such a difference to the method call would not be significant. It is important to remember that this the overhead to the method call, not the method body. The method body is itself likely to execute many statements, most of which will probably not be advised.

For our examples of logging / persisting field writes to disk, this method call overhead would be extremely insignificant compared to the IO required for this. As stated before, Thorn is marketed as a scripting language that can evolve into applications, and is largely used to make distributed programming easier. This should exclude people who would care about the overhead added to advised methods, which leads the author to conclude that the performance of before advice is more than adequate.

After advice gave results almost identical to before advice, as both are implemented in an identical fashion.

5.3.2.3 Method call - around advice

The third benchmark was similar to the second benchmark, with the difference that instead of using before advice, an around advice was used. The around advice first gave the same advice as the before statement, and then used the proceed statement to execute the advised method.

The benchmark suggested a 16 to 30 nanosecond (see table 5.2) overhead of using around advice (most likely due to the proceed call). This is a very low overhead compared to before advice (around 3% higher than before advice), there should be virtually no cases where around advice is not taken where it makes the code clearer.

5.3.2.4 Advising useless entities

The use of aspects does not always add overhead, sometimes using AOP can even reduce overhead while also improving readability.

It is very common for programs to check predicates which do not change throughout the lifetime of the program. For example, the Log4J library uses an external configuration file which determines whether logging is turned on while the application executes, and this predicate is evaluated just before every logging statement.

The solution that some developers utilise is to use a preprocessor, which will remove these predicate checks. Of course, this isn't an ideal solution, it is not possible to change the predicate while the application is running, i.e. to turn on logging at a given time. Furthermore, preprocessors can clutter code, with preprocessor directives near the predicate checks.

Instead, one can simply use aspects. The equivalent of using a preprocessor to remove predicate checks is simply to not instantiate the aspect (perhaps the code which instantiates aspects will check a configuration file, i.e. a single predicate check). The equivalent of switching the behaviour on / off at runtime is to switch an aspect on / off while it is running (for an unselective aspect), or to add / remove objects from the aspect instance (if its a selective aspect).

The author did a benchmark which should motivate some users of AOD-Thorn to use aspects. The benchmark emulates a scenario where a predicate check is done just before every logging statement, as the requirement of the application is that it should be possible to turn it on at some point.

We already benchmarked the overheads of adding advice, however we did not benchmark the overhead of not having advice and doing a predicate check on every method invocation.

The benchmark works by checking how it takes to call and execute a method which contains a predicate check (which itself involves a method call, as would be the case of Log4j, which uses boolean methods such as log.isDebugEnabled()), versus simply not turning the logging aspect on (which, due to the implementation, is performance-wise equivalent of instantiating an object and then removing or switching off the aspect, or using a selective aspect which does not have the logged object added).

The results were extremely encouraging. The version which uses a predicate check upon each invocation took around 8000ns (see table 5.3), versus the 410ns by the solution which used aspect that was not turned on (i.e. unselective aspect which was not instantiated).

Of course, this is most likely due to the slowness of the interpreter, and a compiled version is very unlikely to see such an extreme difference, however there is still likely to be some benefit with the compiled version as well.

Benchmark	Thorn	AOD-Thorn	Mean diff interval[1]	T-test and code
Method call - no advice	328ns	337ns	-3ns to 21ns	Section: B.1
M. call - before advice	568ns	688ns	105ns to 134ns	Section: B.2
M. call - after advice	566ns	685ns	104ns to 133ns	Very similar to B.2

Table 5.1: Thorn vs AOD-Thorn benchmarks.

Benchmark	B./a. advice	Around advice	Mean diff interval[2]	T-test and code
Method call -				
Before / after vs around	676	698	16 to 30	Section: B.2

Table 5.2: Before / after advice vs around advice

Benchmark	No aspects soln.	Aspect solution	Benchmark code
Predicate check vs aspect turned off	8246	411	Section: B.5

Table 5.3: Predicate checks vs aspect turned off

5.3.2.5 Microbenchmarks - local aspecting - conclusion

The micro benchmarks showed us that there was a small constant delay added to joinpoint execution by aspecting when aspect is not giving advice and there is also an overhead of using advice versus a hand written solution. However, the latter benchmark also showed us that the interpreter is actually very slow, for example, a method which contains an extremely simple body, i.e. a method call to a predicate checking method took about 20 times longer to execute than an empty method body.

This gives us excellent evidence to support the claim that any overheads introduced by aspects are extremely insignificant compared to how long a joinpoint takes to execute, which suggests that Thorn developers should not be put off by the aspecting extensions to the language.Having said this, it is also true that the time it takes for the rest of the body to execute compared to the predicate check will also be large, making the predicate check itself insignificant.

For this reason, many of the other combinations possible with aspects were not benchmarked (e.g. lets say field writes with selective aspects), because of the way the algorithmic complexity they introduce is no higher than i.e. method calls with unselective aspects. It was apparent that the results that would be gained from them will be similar to those gained for method interception, i.e. not a cause for developers to be put off. Furthermore, while we may not have much of a performance advantage over hand crafted solutions, hand crafted solutions do not have a performance addition over aspects, and aspects can bring improvements in modularity, readability and the reduction of fragile scattered code.

5.3.3 Microbenchmark - distributed

To get an idea of the type of overhead added by RMI, we ran a simple benchmark (code listing at: B.6) that called a method on a remote aspect instance repeatedly.

On our machine, the average time taken for this method invocation was around 240,000 nanoseconds, or 0.24 milliseconds.

This number may at first seem shocking, that is about 800 times slower than if we didn't use RMI! However, it does mean that the system is capable of over 4000 method invocations a second. This is fast enough for a large number of distributed applications. For example, in our sensors example, it is unlikely that we will need more than 4000 sensor readings every second for most cases.

It should be noted here that network latency also plays a key part in the performance of a distributed network, it is quite possible that the network latency will be the bottleneck rather than the speed of method invocation.

We tried method invocations with up to 5 arguments, which didn't have a noticeable effect on the method invocation speed (remember that we are stubbing arguments, not serialising, so stub creation time is very comparable across different types of arguments).

5.3.3.1 Microbenchmark - distributed - conclusion

We already knew that RMI would put a large overhead on method invocations, although it turned out to be lower than we expected. Based purely on the performance, distributed aspect instances may still be a viable option for some applications, several thousand remote calls per second is good for a large class of problems. This is especially true as the remote aspects are meant as a feature used for prototyping, until the programmer explicitly optimises the code for a distributed architecture.

5.4 Further optimisations

5.4.1 Local aspecting

While it is possible to further increase the speed of local aspecting, given the previous benchmark results, it does not seem necessary to do so.

For example, the execution of joinpoints which are not advised currently has an overhead, as the joinpoint must check if there are any aspects that are advising it. Further, when executing joinpoints with say only before advice, there is also a check for after / around advice. This can be avoided by modifying the joinpoint AST (i.e. the AST would have a node which would cause advice to be evaluated) instead of checking for advice before the evaluation of every joinpoint, including ones which are not advised by any aspects.

This approach is similar to what is done by AOP suites which modify bytecode.

5.4.2 Distributed aspecting

Transparently distributed aspects have several optimisations possible.

1. **Pure functions** - if an aspect method does not involve IO or create a sideeffect (does not mutate any state on the master), then there is no

reason for it to be evaluated on the master aspect. For example, if we have:

```
def addOne(x) = return x+1;

...
azpct.addOne(1);
```

its easy to see that there is no need to ask the master server to carry out this method. However, with the current implementation, it will do so, as the *azpct* keyword was used, which references the master. The burden to evaluate this could be moved to the slave aspect.

2. **RMI** - RMI is synchronous, we must wait for a reply from the remote machine before we continue with our computation. However, we may be able to continue with some computation while we await the remote machine to reply, i.e. turn it into an asynchronous call of sorts. If we apply data flow analysis, we may even be able to execute instructions which are further down the instruction list than the one which requires the reply from the remote machine (i.e. instruction reordering[61], a practise used in microprocessor design as well as e.g. the HotSpot JVM).

3. **Serialisation / stubbing -** some work could be done on more intelligently selecting when to stub an object and when the object graph can be serialised and sent over the wire. This should be a rich area for possible optimisations.

5.5 Scalability

5.5.1 Local aspecting

For this section, we define scalability as the ability of an AOD-Thorn program to utilise aspects more without imposing an overhead which may negate performance requirements of the program.

Aspecting in a local context (i.e. not distributed), appears to have excellent scalability.

5.5.1.1 Program startup time

Program startup time is increased due to the weaving process, where every joinpoint is matched against all pointcuts. The weaving process exhibits a time complexity of $O(n * m)$, where the n stands for number of joinpoints and the m is the number of pointcuts. Of course, how long this weaving process really takes depends on the speed of the regular expression engine.

The experiment described in section A.2 shows that the Java regular expression engine is very fast. 10 million regex matches of two realistic strings took about 1.5 seconds, which would be the time added to program startup if our program had for example 100,000 joinpoints and 100 pointcuts. This appears fast enough for most / all of the applications that have ever been written in Thorn so far.

Nevertheless, if / when application written to be executed by the Thorn interpreter started to get very big (i.e. needing more than 100 million regex matches, which would take about 15 seconds), it would be easy to reduce the time taken by weaving using two simple strategies:

1. Use an incremental compiler[57]. The compilation process would be much like the early Python compilers, which stored a parsed version of a .py file (the AST had certain operations done on it that is always done during program startup). The execution time for this was identical as a non compiled version, but the program startup could avoid having to take certain procedures, making startup faster. The weaving could be part of this compilation process. Furthermore, by using an incremental compiler, there is no need to match all the joinpoints against all the pointcuts when making a change to the application.
2. Speed up the matching process. For some pointcuts, it may not even be required to use a full regular expression engine, i.e. one can use string equals. Further, some kind of tree based data structure could be used so that the pointcuts are only evaluated against those joinpoints which have a chance. For example, if a pointcut fails to evaluate at the package level, then there should be no need to check all the joinpoints in that package later.

Of course, it is unlikely that anyone will ever write a very large program for the Thorn interpreter, so the above ideas would be really be best used in a proper bytecode compiler.

5.5.1.2 Program execution time

When referring to overhead here, we mean the extra time the program has to spend due to using aspects instead of writing the advice near the joinpoint by hand.

Program execution time appears to have a constant time overhead (or if we get very detailed about it, bounded time, due to the way hashmaps work), both in the case of joinpoints with no advice and joinpoints with advice.

Execution of joinpoints with no advice have the constant time overhead of checking whether there is advice for that joinpoint.

Execution of joinpoints with advice has the constant time overhead involving retrieving the advice for the joinpoint.

As such, program execution time should scale very well with larger programs.

5.5.2 Distributed aspecting

The big scalability issue with the distributed aspecting portion of the project is that all the aspect state is held at the master, and it executes all the methods of the aspect. This follows a centralised server architecture, and anyone that knows anything about distributed system will tell you that this is not a good idea.

1. There is no fault tolerance - what happens if the central machine goes down? The entire distributed system stops working.
2. There is no load balancing - all the slaves communicate with one master. Since there is no load balancing, the single machine needs to service all requests. However, load balancing a system like this can be difficult, there is a need to replicate the aspect state, which itself can add overhead, and developing a load balancing mechanism which does not compromise correctness across the whole distributed system can be difficult without experience in this area.

A program not tailored for a distributed setting is likely to be extremely slow. For example, not only do we still have the issue of the interpreter being slow, but we now have the overheads of RMI messaging on the network and the need to create stubs for e.g. arguments and return values, serialising certain values and then finally the network latency (and bandwidth). Further, nothing is done in an asynchronous fashion. Bringing asynchronism into the equation (i.e. by using futures) brings overheads and complexity of its own.

Some of these issues may be tackled, however this seemed a bit out of the scope of the project, given the timelines involved.

5.6 Why AOP faces problems with adoption in industry - how can AOD-Thorn help?

Action at a distance "Action at a distance is an anti-pattern (a recognised common error) in which behaviour in one part of a program varies wildly based on difficult or impossible to identify operations in another part of the program. "[54]. Does AOP exhibit the action at a distance anti-pattern? The answer is probably yes. At the very core, advice is a meta programming construct that changes the behaviour of a program from a different place. After AOP is introduced into a project, when a programmer reads the definition of a joinpoint (such as a method), he can no longer be sure that this is what will be executed when the program runs. Reasoning about the program immediately becomes more difficult.

So what is the solution? Tooling[41]. For example, while editing a pointcutted method, the IDE the programmer uses would tell the programmer that this method is going to be advised at runtime; when the programmer wishes to rename a method, the IDE will tell him whether he has just broken a pointcut somewhere, and therefore the behaviour of this method will change.

AOD-Thorn would be just as vulnerable to the action at a distance anti-pattern as usual AOP suites. The political advantage that AOD-Thorn has, over say Java, is that the aspecting extensions to Thorn would've been made early in the game, before tool vendors had a chance to build tools that were not designed with aspecting in mind. With the aspecting in the core language, the tooling would've had to fully support aspecting in order to be successful.

Lack of standard The lack of standard AOP suite in Java may be another reason for its demise. Again, tooling support is patchy when there is no single standard implementation, something that would've been avoided if Thorn had aspecting extensions to it from the beginning.

Legacy software may be hard to adapt for AOP Sometimes it is a lot of work to get AOP introduced into legacy software. With AOD-Thorn, this would not be a problem, Thorn is still young enough not to have any real large applications.

AOD-Thorn Given the above reasons, putting AOP extensions into the core language may go a long way towards helping AOD-Thorn avoid the problems that other AOP suites for other languages have.

5.7 Transparently distributed aspects limitations

5.7.1 Transparently distributed aspects

5.7.1.1 Are we breaking Thorn rules on distributed computation?

Thorn is a language which uses message passing concurrency but no shared state concurrency. The question arose whether the distributed aspects are breaking the Thorn rules on how distributed applications should be written.

It should be noted, that by default the RMI subsystem used for distributed aspects is in fact a message passing mechanism. During a remote method call, the arguments and return values are usually sent using pass by value semantics[2] (object graphs are serialised). If we had used RMI in this way, it would have worked in exactly the same way as Thorn already does, i.e. object graphs are serialised and then sent over the network.

The transparently distributed aspects idea was a feature aimed as the AOP mechanism to be used straight after a local application is turned into a distributed application, i.e. for experiments on a distributed architecture, rather than as a full replacement for a solution that gave the programmer explicit control over what is sent over the network.

In Thorn, there is a "pure" annotation for classes, which indicates that the class "cannot refer to mutable or component-local state, and thus, its instances are suitable for transmission between components"[6]. If a class was not marked as pure (because the program was not distributed, even though it is a good candidate the purity status) and we just moved to a distributed setting, we could not use objects of that class when communicating with remote aspects, they will simply not serialise (this is enforced in the code). This would prove problematic: the programmer has to do extra work for the distributed aspects which he wasn't expecting.

Furthermore, the author felt it was unlikely that during the initial stages of moving a single machine program to a distributed architecture, the object graphs would've been optimised for efficient serialisation (some objects may be part of large object graphs). This is why the transparently distributed aspects use pass by value semantics.

The reader may recall that during the advice of execution, when we ask a remote machine to do something (e.g. advice being executed by the slave has a call to the master), the local machine stops executing the program. In this sense, there is no concurrent computation taking place (across the distributed system), so the problem that message passing is trying to solve does not exist here.

Its also worth noting here that the RMI subsystem will queue up requests for method invocations (i.e. if slave b wanted to invoke a method on the master while it was carrying out something for slave a), it will not execute them at the same time. However, in implemented version, there does seem to be a problem with interleaved advice executions, i.e. the advice for slave A may have multiple invocations on the master, however in between invocations, slave B may put in a request. This problem, along with the whole idea of distributed aspects needs a much more formal treatment to work out such details.

5.7.1.2 IO

One of the big problems with the transparently distributed aspects is to do with IO.

An application that moved from a single machine to a distributed architecture that used IO may break, (in the example of file IO) a file that would've been created and then read by the application may no longer exist in the distributed manner (e.g. the master would create the file, and the slave would attempt to read it). This may mean work for the programmer, i.e. the *transparent* aspects would not be that transparent anymore.

Problems like this may be solved by the interpreter acting as a type of sandbox environment, where for example, file IO is intercepted and redirected to some type of virtual, distributed file system.

5.7.1.3 The essence of transparently distributed aspects

On hindsight, the introduction of transparently distributed aspect to the language was in fact the introduction of a more general idea, remote method invocation.

However, the way we implemented them emulates a special case of RMI in Java: in Java RMI we need to take special steps in order to be able to export an object (i.e. it needs to extend an RMI interface and methods need to be defined as throwing a RemoteException), however, in our language, we used RMI with pass-by-reference semantics by default (as opposed to pass by value) and every object is exportable, with no special work required for this.

As such, it may have been better to explore the idea of introducing RMI to not only aspect instances, but to all objects in Thorn. However, such a remote method invocation mechanism already exists, although it is restricted to components, not objects in general. Furthermore, components use raw sockets, which is a questionable (low level) technique when much higher level mechanisms are available.

5.8 Correctness

As the future of Thorn was somewhat uncertain from the beginning of the project, correctness of implementation has not been the highest priority throughout the project. Instead, the author decided to use the Thorn interpreter as a testbed for his ideas, however, unit testing was nevertheless used throughout the project as a means of adding at least some confidence in the correctness of the project.

There were extensive tests that came with the interpreter at the beginning of the project, and all the tests that passed at the start of the project still passed at the end (and some which did not pass at the beginning were made to pass), to which the new ones were added.

Sidenote It is likely that the reason some of the unit tests are failing, is that previous developers of Thorn did not run the unit tests continuously, they do take a long time to complete. It would be recommended that any future developers use a continuous integration tool such as Hudson, which will monitor the version control (SVN) for commits, and will automatically run the unit tests on the CI server (and notify developers if unit tests fail). This should ensure that proper testing hygiene is adhered to, avoiding the problem of future developers testing less as a result of previous failures.

Chapter 6

Conclusions

In this project we have explored the addition of the aspect-oriented paradigm into Thorn.

We added features which are common to many AOP suites such as AspectJ, including lexical pointcut descriptors using full regular expressions, before/after-/around advice with type checking on the arguments, all of which works with field reads and writes as well as method invocations.

We explored and implemented the idea of aspects which are instantiated by the user instead of the AOP suite instantiating it for the programmer. Aspects can be both selective and unselective, instead of placing this burden on the programmer.

Furthermore, the idea of aspect instances operating on a remote machine was explored and implemented, although not to the same maturity as the rest of the work.

As with other AOP suites, the AOP extensions we brought to Thorn have great potential to reduce scattering and tangling in real, non-trivial applications. The feature set is rich enough to use many of the tangling / scattering reduction techniques used in other languages with mature AOP suites.

The performance hit we added as a result of the extensions appears to be irrelevant for local aspect instances, and usable for some applications with remote aspects.

Of course, our extension suffers from the same problem as all AOP implementations, namely the heavy reliance on tooling for it to be a viable paradigm used in non-trivial software projects.

A lot of work remains to be done before our extension for Thorn, or indeed Thorn itself, become viable for real, non-trivial software projects. We already presented much of this in descriptions of limitations of our work, but we present some more below.

6.1 Future work

- The Thorn language has no real published formal semantics. A useful piece of work would be to formally write the semantics of the language. This base could then be used for descriptions of any extensions to the language, such as the extensions that were made in this project. While we presented some semantics in the implementation chapter, these were not exact enough, as there were no base semantics.
- The idea of distributing aspects needs a full formal specification, to which the implementation can be done. It should deal with the problems of keeping the implementation performant while not breaking the semantics of the language, especially for issues such as IO and the interleaving of advices.
- Many dynamically typed languages have AOP suites, however, it has been suggested that AOP is not as useful for this class of languages as it is for static languages such as Java. There does not seem to be much evidence of AOP use in languages with gradual typing systems as in Thorn. It may be useful to see how the implementation of AOP suites can exploit the properties of gradual typing, it at all.
- There are opportunities to apply static analysis methods in order to determine an optimised strategy to distribute aspects. In other words, some work could be done to make intelligent decisions about how communication between aspect instances is done, i.e. when to do something at the slave and when to do it at the master.
- When the field of a master aspect instance is overwritten, this is not broadcasted to the slaves. It may be possible to write an optimisation for the distributed aspects which may move computation to the slave, if the master broadcasts events when a field is written to, in a replication mechanism. Based on this, slaves may be able to execute advice without having to ask the master to do this, even if the advice depends on the master aspect state. Some problems may include atomic broadcasts from the master to slaves, etc.
- A Thorn compiler was supposed to have been released, although by the looks of it, it will not be. However, if it ever did resurface, a good piece of work may be to implement an AOP suite similar to the one in AOD-Thorn into the compiled version. Of course, the compiler may very well have different performance requirements than the interpreter does.
- As discussed in the background section, dynamic weaving (weaving advice after the program has started) is now seen as a useful feature for AOP suites. While we began working on this, unfortunately we ran out of time and did not finish it. It does not seem too difficult to pull off, especially in the interpreter. One only has to modify ASTs, with the difficult bit being

security surrounding the delivery of an AST at runtime (perhaps by using public key cryptography).

- Thorn currently has almost no API outside the core language constructs. It may be worth working on a set of wrappers for the Java API as well as Thorn specific ones, in order to enable the language to be used for non toy projects.
- While Java is starting to be seen as a dinosaur by some practitioners, the JVM itself is very much still an incredible piece of engineering. New languages are joining the JVM all the time, and there is now a trend for polyglot projects, spanning languages such as Scala, Clojure, groovy and so on. An exciting area to explore may be to build a language agnostic AOP implementation, i.e. one that could give advice across language boundaries. For example, Scala can almost seamlessly use Java code, so it would be interesting to see if one could build an AOP framework where e.g. Scala code advised Java code or vice versa.

Appendix A

Listings

A.1 AspectJ experiment to show aspect instantiation behaviour

Listing A.1: Clazz.java

```
package com.company.employeebeans;

public class Clazz {

  public static void main(String[] args){
    System.out.println("Just before Clazz.m()");
    m();
  }

  private static void m(){
    System.out.println("inside Clazz.m()");
  }

}
```

Listing A.2: Asp.aj

```
package com.company.employeebeans;

public aspect Asp {

  public Asp(){
    System.out.println("Asp constructor");
  }

  after(): call(void Clazz.m(..)){
    System.out.println("Asp advice after clazz.m()");
  }

}
```

The above listings result in the following output on the console:

```
Just before Clazz.m()
Asp constructor
inside Clazz.m()
Asp advice after clazz.m()
```

This is the behaviour we would expect if we had a class of the form (i.e. lazy loading):

```
class C{
  static { C(); }
}
```

A.2 Java experiment to measure speed of the regex engine

```
import java.util.regex.Pattern;

public class Main {
  public static void main(String[] args) {
    long before = System.nanoTime();
    int iterations =10000000;
    Pattern pattern = Pattern.compile("com.mypackage.set.*");
    for(int i = 0; i < iterations; i++){
      pattern.matcher("com.myspackage.setHi").matches();
    }
    System.out.println(("Time taken: "+(System.nanoTime() - before)));
  }
}
```

Time taken on our machine: 1.5 seconds

Appendix B

Benchmarks and t-test results

B.1 Method execution - no advice

B.1.1 Paired T-test result

```
P value and statistical significance:
  The two-tailed P value equals 0.1141
  Not statistically significant

Confidence interval:
  The mean of AOD-Thorn minus Thorn equals 8.99
  95% confidence interval of this difference:
  From -2.702 to 20.689
```

B.1.2 Benchmark

Benchmark was ran on both Thorn and AOD-Torn interpreter

```
class C{
  def m() = {};
}


obj = C();

inneriterations = 20000000;
outeriterations = 10;
for(i <- 1..outeriterations){
```

```


var timeBefore := nanoTime();
for(n <- 1..inneriterations){
 obj.m();
}
var timeAfter := nanoTime();
println((timeAfter - timeBefore)/inneriterations);
}
```

B.2 Method execution - before / after advice

B.2.1 Paired T-test results

```
P value and statistical significance:
  The two-tailed P value is less than 0.0001
  Statistically extremely significant

Confidence interval:
  The mean of AOD-Thorn minus Thorn equals 119.47
  95% confidence interval of this difference: From 104.94 to 134
```

B.2.2 Benchmark ran on Thorn interpreter

```
class C{
  def m() = {this.m1();};
  def m1() = {};
}


obj = C();

outeriterations = 10;
inneriterations = 2000000;
for(i <- 1..outeriterations){

var timeBefore := nanoTime();
for(n <- 1..inneriterations){
 obj.m();
}
var timeAfter := nanoTime();

println((timeAfter-timeBefore) / inneriterations);
}
```

B.3 Method execution - before / after vs around advice

B.3.1 Paired T-test results

```
P value and statistical significance:
  The two-tailed P value is less than 0.0001
  By conventional criteria ,
  this difference is considered to be
  extremely statistically significant.

Confidence interval:
  The mean of Around advice minus B/a advice equals 22.96
  95% confidence interval of this difference: From 16.45 to 29.48
```

B.3.2 Benchmark code

The benchmark ran for before / after advice was as in section B.2.

The benchmark involved using the same method.th as the before / after benchmark, however the aspect code was modified:

Listing B.1: aspect.th

```
module aspects{
  aspect unselective Aspectc{
   around: exec jp[(m)] args(){
     this.m1();
     proceed();
   }
  }
}
```

B.4 Benchmark ran on AOD-Thorn interpreter

Listing B.2: method.th

```
import aspects.*;
class C{
  def m() = {};
  def m1() = {};
}
readln();

Aspectc();
```

```
obj = C();

outeriterations = 10;
inneriterations = 2000000;
for(i <- 1..outeriterations){

var timeBefore := nanoTime();
for(n <- 1..inneriterations){
 obj.m();
}
var timeAfter := nanoTime();

println((timeAfter-timeBefore) / inneriterations);
}
```

Listing B.3: aspect.th

```
module aspects{

  aspect unselective Aspectc{
   before: exec jp[(m)] args(){
     this.m1();
   }
  }
}
```

B.5 Benchmark - aspect vs predicate check

B.5.1 Using a predicate check

```
class Logger{
  // normally set by a config
  // file
  val turnedOn = false;

  def shouldLog(){
    return turnedOn;
  }
}

class C{
  val log = Logger();

  def m() = {
    if(this.log.shouldLog()){
      //skip
    }
  };

}


obj = C();

outeriterations = 10;
inneriterations = 2000000;
for(i <- 1..outeriterations){

var timeBefore := nanoTime();
for(n <- 1..inneriterations){
 obj.m();
}
var timeAfter := nanoTime();

println((timeAfter-timeBefore) / inneriterations);
}
```

B.5.2 Using aspects

Same aspect code as before.

```
import aspects.*;
class Logger{
  // normally set by a config
  // file
  val turnedOn = false;

  def shouldLog(){
    return turnedOn;
  }
}

class C{
  def m() = {};
}

obj = C();

outeriterations = 10;
inneriterations = 2000000;
for(i <- 1..outeriterations){
```

```

var timeBefore := nanoTime();
for(n <- 1..inneriterations){
 obj.m();
}
var timeAfter := nanoTime();

println((timeAfter-timeBefore) / inneriterations);
}
```

B.6 Benchmark - distributed method invocation

Listing B.4: aspect.th

```
module Aspects{
 aspect selective Aspect{
    def m() = {}

    before: exec jp[start] args(){

        outeriterations = 10;
        inneriterations = 20000;
        for(i <- 1..outeriterations){

        var timeBefore := nanoTime();
        for(n <- 1..inneriterations){
         azpct.m();
        }
        var timeAfter := nanoTime();

        println((timeAfter-timeBefore) / inneriterations);
      }
    }
  }
}
```

Listing B.5: master.th

```
import Aspects.*;
asp = Aspect();
masterAspect(asp, "ASP");

println("Master running");
```

Listing B.6: slave.th

```
import Aspects.*;
class C{
  def start() = {}
}

c = C();
asp = slaveAspect("localhost", "ASP");
addToAsp(c, asp);

c.start();
```

Appendix C

List of changes

The following is the list of files changed. An “A” in the first column represents a file which was added, and an “M” represents a file that was modified. It was generated using the “SVN status” command. Note that most of the files in the package fisher.syn (although not all, i.e. fisher.syn.core) are generated by the src/syntax/ast-fisher.py script (this script was modified).

```
A fisher/runtime/AspectDynamizer.java
A fisher/runtime/RemoteSyntax.java
A fisher/runtime/MethodInvocationRequest.java
A fisher/runtime/RemoteThing.java
A fisher/runtime/AspectObject.java
A fisher/runtime/ObjectRemoteIface.java
A fisher/runtime/AspectDynamicTh.java
A fisher/runtime/RemoteAspectObject.java
A fisher/runtime/builtInFun/MasterAspectBIF.java
A fisher/runtime/builtInFun/DelFromAspBIF.java
A fisher/runtime/builtInFun/AddToAspBIF.java
A fisher/runtime/builtInFun/NanoTimeBIF.java
A fisher/runtime/builtInFun/SlaveAspectBIF.java
M fisher/runtime/builtInFun/PrintlnBIF.java
M fisher/runtime/Nullity.java
M fisher/runtime/ObjectTh.java
M fisher/runtime/StringTh.java
M fisher/runtime/Internal_SortableTh.java
M fisher/runtime/IntRangeTh.java
M fisher/runtime/dunno/StringTh_16.java
M fisher/runtime/dunno/StringTh_17.java
M fisher/runtime/ListTh.java
M fisher/runtime/BuiltInFunctionTh.java
M fisher/runtime/CharTh.java
```

```
M fisher/runtime/Thing.java
M fisher/runtime/FileTh.java
M fisher/runtime/OrdTh.java
M fisher/runtime/JavalyFunImpl.java
M fisher/runtime/TableTh.java
M fisher/runtime/dist/AbstractLetter.java
M fisher/runtime/dist/MsgComp2Comp.java
M fisher/runtime/dist/SiteData.java
M fisher/runtime/ModuleDynamicTh.java
M fisher/runtime/IntTh.java
M fisher/runtime/lib/socketeer/MsgComp2StringSocket.java
M fisher/runtime/lib/json/JSON.java
M fisher/runtime/lib/sox/SoxUtil.java
M fisher/runtime/lib/sox/NetSite.java
M fisher/runtime/SetTh.java
M fisher/runtime/ThingExtended.java
M fisher/runtime/SiteTh.java
M fisher/runtime/FloatTh.java
M fisher/runtime/ClassDynamicTh.java
M fisher/runtime/ClosureTh.java
M fisher/runtime/RecordTh.java
M fisher/runtime/DirTh.java
M fisher/runtime/HttpTh.java
M fisher/runtime/AbstractRangeTh.java
M fisher/runtime/BytesTh.java
M fisher/runtime/BoolTh.java
M fisher/runtime/ComponentTh.java
M fisher/run/REPL.java
M fisher/run/Thorn.java
A fisher/eval/AspectObjectRegistry.java
A fisher/eval/AspectRegistryFieldAccessMatcher.java
A fisher/eval/AspectRegistry.java
A fisher/eval/AspectRegistryMethodCallMatcher.java
A fisher/eval/AspectRegistryPctctMatcher.java
A fisher/eval/AspectsComputer.java
A fisher/eval/FieldAccessAspectRegistry.java
M fisher/eval/interfaces/Framelike.java
M fisher/eval/EvalUtil.java
M fisher/eval/ModuleDynamizer.java
M fisher/eval/Evaller.java
M fisher/eval/Frame.java
M fisher/eval/Computer.java
A fisher/test/Regex.java
A fisher/test/AOPTest.java
M fisher/test/EvalTest.java
M fisher/test/AllTests.java
```

```
M fisher/test/ParserTest.java
A src/fisher/syn/AdviceSpec.java
A src/fisher/syn/AspectExtends.java
A src/fisher/syn/AspectDecl.java
A src/fisher/syn/Proceed.java
A src/fisher/syn/Pointcut.java
A src/fisher/syn/Slave.java
A src/fisher/syn/Azpct.java
A src/fisher/syn/AdviceDecl.java
A src/fisher/syn/AspectPointcut.java
A src/fisher/syn/Origin.java
A src/fisher/syn/AdvicePtctPrimitiv.java
A src/fisher/syn/Aspect.java
A src/fisher/syn/Joinpoint.java
A src/fisher/syn/AspectsInAList.java
M src/fisher/syn/AssignToSubscripted.java
M src/fisher/syn/TableKey.java
M src/fisher/syn/PatNot.java
M src/fisher/syn/PatListBitEllip.java
M src/fisher/syn/ClsPatDef.java
M src/fisher/syn/QueryQuantifierCount.java
M src/fisher/syn/QuerySort.java
M src/fisher/syn/QueryControlFor.java
M src/fisher/syn/ModuleFileImport.java
M src/fisher/syn/VarDecl.java
M src/fisher/syn/QueryQuantifierSome.java
M src/fisher/syn/RecordField.java
M src/fisher/syn/Continue.java
M src/fisher/syn/AssignToMap.java
M src/fisher/syn/MethDecl.java
M src/fisher/syn/Javaly.java
M src/fisher/syn/ProcInit.java
M src/fisher/syn/ListBitExp.java
M src/fisher/syn/Alias.java
M src/fisher/syn/AssignTarget.java
M src/fisher/syn/OpABExp.java
M src/fisher/syn/DotMethodCallExp.java
M src/fisher/syn/ListBitEllip.java
M src/fisher/syn/TypedExp.java
M src/fisher/syn/PatEvalTestExp.java
M src/fisher/syn/ComparisonBit.java
M src/fisher/syn/PostExpRecordArgs.java
M src/fisher/syn/TableFields.java
M src/fisher/syn/PatListBit.java
M src/fisher/syn/PatExtract.java
M src/fisher/syn/QueryFirstlike.java
```

```
M src/fisher/syn/SetCtor.java
M src/fisher/syn/PatVar.java
M src/fisher/syn/HighLevelCommunication.java
M src/fisher/syn/StringWithInterpolations.java
M src/fisher/syn/FunBody.java
M src/fisher/syn/RecordCtor.java
M src/fisher/syn/WildcardExp.java
M src/fisher/syn/Probe.java
M src/fisher/syn/LabellableLoop.java
M src/fisher/syn/MatchExp.java
M src/fisher/syn/StringBitText.java
M src/fisher/syn/Case.java
M src/fisher/syn/ClsExtends.java
M src/fisher/syn/QueryFirst.java
M src/fisher/syn/QueryTable.java
M src/fisher/syn/ProcBody.java
M src/fisher/syn/EvalTestExpExp.java
M src/fisher/syn/Serve.java
M src/fisher/syn/SyncStmt.java
M src/fisher/syn/AsyncStmt.java
M src/fisher/syn/PatTypeTest.java
M src/fisher/syn/PatRecordField.java
M src/fisher/syn/ExpExtract.java
M src/fisher/syn/JavalyNewDecl.java
M src/fisher/syn/SuperCall.java
M src/fisher/syn/PatWildcard.java
M src/fisher/syn/ModuleFileAlias.java
M src/fisher/syn/PatLiteral.java
M src/fisher/syn/Formals.java
M src/fisher/syn/ListCtor.java
M src/fisher/syn/MapCtor.java
M src/fisher/syn/BracketCall.java
M src/fisher/syn/SpawnByComponentName.java
M src/fisher/syn/QueryQuantifierEvery.java
M src/fisher/syn/While.java
M src/fisher/syn/ListBit.java
M src/fisher/syn/PostExpBracketArgs.java
M src/fisher/syn/StringBitVar.java
M src/fisher/syn/Try.java
M src/fisher/syn/QGKey.java
M src/fisher/syn/SuperCtorCall.java
M src/fisher/syn/Assign.java
M src/fisher/syn/QueryControlIf.java
M src/fisher/syn/PatOr.java
M src/fisher/syn/For.java
M src/fisher/syn/PatMatchSomethingElse.java
```

```
M src/fisher/syn/Send.java
M src/fisher/syn/QueryListComprehension.java
M src/fisher/syn/AssignTofield.java
M src/fisher/syn/QueryQuantifier.java
M src/fisher/syn/QuerySwiss.java
M src/fisher/syn/Recv.java
M src/fisher/syn/PatNotNull.java
M src/fisher/syn/SortKey.java
M src/fisher/syn/QualName.java
M src/fisher/syn/Cmd.java
M src/fisher/syn/Throw.java
A src/fisher/syn/interfaces/AspectMember.java
M src/fisher/syn/SyncDecl.java
M src/fisher/syn/AsyncDecl.java
M src/fisher/syn/QueryControl.java
M src/fisher/syn/Skip.java
M src/fisher/syn/SuperThingie.java
M src/fisher/syn/RecordCall.java
M src/fisher/syn/SyntaxInAList.java
M src/fisher/syn/QuerySetComprehension.java
M src/fisher/syn/QueryControlVar.java
M src/fisher/syn/PatSlash.java
M src/fisher/syn/FunDecl.java
M src/fisher/syn/PatMethodCall.java
M src/fisher/syn/Module.java
M src/fisher/syn/QueryAfter.java
M src/fisher/syn/ClsCtorDef.java
M src/fisher/syn/AssignToFieldOfSubscripted.java
M src/fisher/syn/QueryAbstract.java
M src/fisher/syn/AbstractStringBit.java
M src/fisher/syn/Literal.java
M src/fisher/syn/JavalyFun.java
M src/fisher/syn/ModArgBinding.java
M src/fisher/syn/CmdsInAList.java
M src/fisher/syn/AbstractTable.java
M src/fisher/syn/Break.java
M src/fisher/syn/TypeConstraint.java
M src/fisher/syn/Seq.java
M src/fisher/syn/OpExp.java
M src/fisher/syn/PatListBitExp.java
M src/fisher/syn/PatInterpolation.java
M src/fisher/syn/ModulesInAList.java
M src/fisher/syn/If.java
M src/fisher/syn/QueryControlWhile.java
M src/fisher/syn/IdWithOptInit.java
M src/fisher/syn/AnonFun.java
```

```
M src/fisher/syn/AssignToId.java
M src/fisher/syn/InterpolationExp.java
M src/fisher/syn/ClassFormal.java
M src/fisher/syn/PatAnd.java
M src/fisher/syn/This.java
M src/fisher/syn/Valof.java
M src/fisher/syn/PatListCtor.java
A src/fisher/syn/core/AdvicePtctPrimitive.java
A src/fisher/syn/core/AdviceLocation.java
A src/fisher/syn/core/AdviceAction.java
M src/fisher/syn/core/ComparisonOp.java
M src/fisher/syn/core/Op.java
M src/fisher/syn/QueryControlVal.java
M src/fisher/syn/ClsDecl.java
M src/fisher/syn/PostExpArgs.java
M src/fisher/syn/Return.java
M src/fisher/syn/QGAccum.java
M src/fisher/syn/QueryGroup.java
M src/fisher/syn/Ord.java
M src/fisher/syn/FunCall.java
M src/fisher/syn/PostExpDotId.java
M src/fisher/syn/AnonObj.java
M src/fisher/syn/ForPatternOnly.java
M src/fisher/syn/Table.java
M src/fisher/syn/Parens.java
M src/fisher/syn/Spawn.java
M src/fisher/syn/ComponentDecl.java
M src/fisher/syn/Match.java
M src/fisher/syn/ModuleFileVisibility.java
M src/fisher/syn/ListForGroup.java
M src/fisher/syn/ServeBlock.java
M src/fisher/syn/PatSet.java
M src/fisher/syn/PatRange.java
M src/fisher/syn/Signature.java
M src/fisher/syn/PostExp.java
M src/fisher/syn/JavalyClassDecl.java
M src/fisher/syn/VarExp.java
M src/fisher/syn/TypeConstraints.java
M src/fisher/syn/Comparison.java
M src/fisher/syn/PatRecordCtor.java
M src/fisher/syn/Bind.java
M src/fisher/syn/MethodCall.java
M src/fisher/syn/JavalyMethodDecl.java
A src/fisher/syn/visitor/AspectMemberWalker.java
A src/fisher/syn/visitor/AspectMemberVisitor.java
M src/fisher/syn/visitor/LocalMemberVisitor.java
```

```
M src/fisher/syn/visitor/VisitPat.java
M src/fisher/syn/visitor/ClassMemberWalker.java
M src/fisher/syn/visitor/VisitAssignTarget.java
M src/fisher/syn/visitor/ProcMemberVisitor.java
M src/fisher/syn/visitor/VisitCmd.java
M src/fisher/syn/visitor/ComponentInfoWalker.java
M src/fisher/syn/visitor/PureticVisitor.java
M src/fisher/syn/visitor/VanillaWalker.java
M src/fisher/syn/visitor/ModuleFileMemberVisitor.java
M src/fisher/syn/visitor/ComponentInfoVisitor.java
M src/fisher/syn/visitor/TableMemberVisitor.java
M src/fisher/syn/visitor/Visitor.java
M src/fisher/syn/visitor/ProcMemberWalker.java
M src/fisher/syn/visitor/WalkPat.java
M src/fisher/syn/visitor/ObjectMemberWalker.java
M src/fisher/syn/visitor/VanillaVisitor.java
M src/fisher/syn/visitor/VisitQueryAbstract.java
M src/fisher/syn/visitor/VanillaWalkPat.java
M src/fisher/syn/visitor/VanillaVisitPat.java
M src/fisher/syn/visitor/ClassMemberVisitor.java
M src/fisher/syn/visitor/ObjectMemberVisitor.java
M src/fisher/syn/visitor/ClasslikeWalker.java
M src/fisher/syn/visitor/PureticWalker.java
M src/fisher/syn/visitor/ModuleFileMemberWalker.java
M src/fisher/syn/visitor/VisitPat2.java
M src/fisher/syn/visitor/TableMemberWalker.java
M src/fisher/syn/visitor/VisitQueryControl.java
M src/fisher/syn/visitor/Walker.java
M src/fisher/syn/visitor/LocalMemberWalker.java
M src/fisher/syn/visitor/VanillaVisitCmd.java
M src/fisher/syn/visitor/ClasslikeVisitor.java
M src/fisher/syn/FieldRef.java
M src/fisher/syn/ImportStmt.java
M src/fisher/syn/Pat.java
M src/fisher/syn/ModuleFileMemberStmt.java
M src/fisher/syn/MonoBody.java
M src/fisher/syn/ItExp.java
M src/fisher/syn/queries/SynUtil.java
A fisher/statics/ExtractSealsFromAspectDecl.java
A fisher/statics/AspectProcessor.java
A fisher/statics/SealForAspect.java
A fisher/statics/AspectStatic.java
M fisher/statics/SealMaker.java
M fisher/statics/purity/StaticPurityChecker.java
M fisher/statics/ClassStatic.java
M fisher/statics/Seal.java
```

```
M fisher/statics/ExtractSealsFromModuleMember.java
M fisher/statics/Sealant.java
M fisher/statics/Env.java
M fisher/statics/SealKind.java
M fisher/statics/PredefinedIdentifiers.java
A fisher/parser/grammar-fisher.html
M fisher/parser/TokenMgrError.java
M fisher/parser/SimpleCharStream.java
M fisher/parser/Token.java
M fisher/parser/grammar-fisher.jj
M fisher/parser/ParseException.java
M fisher/parser/FisherParserTokenManager.java
M fisher/parser/FisherParserConstants.java
M fisher/parser/FisherParser.java
M fisher/util/SpecialCharacters.java
M fisher/util/Doom.java
M syntax/ast-fisher.py
```

Bibliography

[1] Java programming dynamics - developerworks series. `http://www.ibm.com/developerworks/java/library/j-dyn0429/`.

[2] jguru: Remote method invocation (rmi). `http://java.sun.com/developer/onlineTraining/rmi/RMI.html` - accessed 18/05/2011.

[3] Columbia University Alexander V. Konstantinou. Java rmi : Remote method invocation - lecture notes on computer networks. Spring 2000.

[4] Danilo Ansaloni, Walter Binder, Alex Villazón, and Philippe Moret. Rapid development of extensible profilers for the java virtual machine with aspect-oriented programming. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 57–62, New York, NY, USA, 2010. ACM.

[5] Remko Bijker. Performance effects of aspect oriented programming.

[6] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the jvm. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 117–136, New York, NY, USA, 2009. ACM.

[7] Silvia Breu, Thomas Zimmermann, and Christian Lindig. Aspect mining for large systems. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 641–642, New York, NY, USA, 2006. ACM.

[8] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[9] Sandra I. Casas, J. Baltasar, Garcia Perez-schofield, and Claudia A. Marcos. Associations in conflict, 2007.

[10] Andy Clement. Aspect-oriented programming with ajdt. In *In ECOOP Workshop on Analysis of Aspect-Oriented Software*. Springer, 2003.

[11] FabrÃcio de Alexandria Fernandes and Thais Batista. Dynamic aspect-oriented programming: an interpreted approach, 2004.

[12] Douglas R. Dechow. Advanced separation of concerns for dynamic, lightweight languages. In *In: 5 th Generative Programming and Component Engineering*, 2003.

[13] E. W. Dijkstra. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[14] Simplice Djoko Djoko, Rémi Douence, Pascal Fradet, and Didier Le Botlan. Casb: Common aspect semantics base. Technical Report AOSD-Europe Deliverable D41, AOSD-Europe-INRIA-7, INRIA, France, 10 February 2006 2006.

[15] Prof. Sophia Drossopoulou. L2 - a formal, minimal, imperative, class based, object-oriented language with inheritance, without over-loading.

[16] E. Figueiredo, B. Silva, C. Sant'Anna, A. Garcia, J. Whittle, and D. Nunes. Crosscutting patterns and design stability: An exploratory analysis. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 138 –147, may 2009.

[17] Ryan M. Golbeck, Samuel Davis, Immad Naseer, Igor Ostrovsky, and Gregor Kiczales. Lightweight virtual machine support for aspectj. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 180–190, New York, NY, USA, 2008. ACM.

[18] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 161–173, New York, NY, USA, 2002. ACM.

[19] Erik Hilsdale and Jim 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.

[20] Shan Shan Huang and Yannis Smaragdakis. Easy language extension with meta-aspectj. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 865–868, New York, NY, USA, 2006. ACM.

[21] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[22] Antti Kervinen. aspects.py - a lightweight approach to aspect-oriented programming in python. `http://www.cs.tut.fi/~ask/aspects/` - accessed 14/05/2011.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[25] E. Kühn, G. Fessl, and F. Schmied. Aspect-oriented programming with runtime-generated subclass proxies and .net dynamic methods. *Journal of .NET Technologies*, 4:1801–2108, 2006.

[26] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[27] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*, pages 121 – 123. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[28] Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 1–12, New York, NY, USA, 2003. ACM.

[29] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *IN PROCEEDINGS OF THE 22ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, pages 418–427. ACM Press, 2000.

[30] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Methodol.*, 17:3:1–3:37, December 2007.

[31] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, MACS '05, pages 1–5, New York, NY, USA, 2005. ACM.

[32] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[33] Marcin Lech Matusiak. aopy: aspect-oriented python. `https://github.com/numerodix/aopy` - accessed 14/05/2011, 2010.

[34] Philipp Haller Michel Schinz. *A Scala Tutorial for Java programmers - page 3*, 1.3 edition, May 2011.

[35] Russell Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[36] Doug Orleans. Incremental programming with extensible decisions. In *In Proceedings of the 1st international conference on Aspect-oriented software development*, pages 56–64. ACM Press, 2002.

[37] Ignacio Solla Paula. Jcthorn: Extending thorn with joins and chords. Master's thesis, Imperial College London, 2010.

[38] Renaud Pawlak, Jean-Philippe Retaillé, and Lionel Seinturier. *Foundations of AOP for J2EE Development (Foundation).* Apress, Berkely, CA, USA, 2005.

[39] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE'05*, pages 59–68, 2005.

[40] Hridesh Rajan and Kevin J. Sullivan. Unifying aspect- and object-oriented design. *ACM Trans. Softw. Eng. Methodol.*, 19:3:1–3:41, August 2009.

[41] Daniel Sabbah. Aspects: from promise to reality. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 1–2, New York, NY, USA, 2004. ACM.

[42] Vladimir O. Safonov. *Using Aspect-Oriented Programming for Trustworthy Software Development.* Wiley-Interscience, New York, NY, USA, 2008.

[43] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *IN: SECOND INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING (GPCE03*, pages 189–208. Springer, 2003.

[44] Peri Tarr, Harold Ossher, William Harrison, Stanley M. Sutton, and Jr. N degrees of separation: Multi-dimensional separation of concerns. pages 107–119, 1999.

[45] AspectJ team. Aspectj - semantics. `http://www.eclipse.org/aspectj/doc/released/progguide/semantics-aspects.html`.

[46] AspectJ team. Frequently asked questions about aspectj. `http://www.eclipse.org/aspectj/doc/released/faq.html`, 2006.

[47] AspectWerkz team. Aop benchmark - comparison of aop suites for java. `http://docs.codehaus.org/display/AW/AOP+Benchmark`, 2004.

[48] Spring AOP team. Spring aop: Aspect oriented programming with spring. `http://static.springsource.org/spring/docs/1.2.x/reference/aop.html` - accessed 19 June 2011.

[49] The JBossAOP team. *JBossAOP User Guide*, 2.0 edition.

[50] The AspectJ Team. The AspectJ Project. URL: http://www.eclipse.org/aspectj/. Retrieved 25/05/2011.

[51] Robin van der Rijst, Marius Marin, and Arie van Deursen. Sort-based refactoring of crosscutting concerns to aspects. In *Proceedings of the 2008 AOSD workshop on Linking aspect technology and evolution*, LATE '08, pages 4:1–4:5, New York, NY, USA, 2008. ACM.

[52] Santiago Vidal, Esteban S. Abait, Claudia Marcos, Sandra Casas, and J. Andrés Díaz Pace. Aspect mining meets rule-based refactoring. In *Proceedings of the 1st workshop on Linking aspect technology and evolution*, PLATE '09, pages 23–27, New York, NY, USA, 2009. ACM.

[53] Daniel Wiese, Uwe Hohenstein, and Regine Meunier. How to convince industry of aop. In *In Proc. of Industry Track at AOSD 2007*, 2007.

[54] Wikipedia. Action at a distance (computer programming) — wikipedia, the free encyclopedia, 2010. [Online; accessed 19-June-2011].

[55] Wikipedia. Business object — wikipedia, the free encyclopedia, 2010. [Online; accessed 2-June-2011].

[56] Wikipedia. Pointcut — wikipedia, the free encyclopedia, 2010. [Online; accessed 23-May-2011].

[57] Wikipedia. Incremental compiler — wikipedia, the free encyclopedia, 2011. [Online; accessed 19-June-2011].

[58] Wikipedia. Java remote method invocation — wikipedia, the free encyclopedia, 2011. [Online; accessed 11-June-2011].

[59] Wikipedia. Join point — wikipedia, the free encyclopedia, 2011. [Online; accessed 23-May-2011].

[60] Wikipedia. Metaclass — wikipedia, the free encyclopedia, 2011. [Online; accessed 27-May-2011].

[61] Wikipedia. Out-of-order execution — wikipedia, the free encyclopedia, 2011. [Online; accessed 20-June-2011].

[62] Wikipedia. Sql injection — wikipedia, the free encyclopedia, 2011. [Online; accessed 18-June-2011].

[63] Jan Wloka, Robert Hirschfeld, and Joachim Hänsel. Tool-supported refactoring of aspect-oriented programs. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 132–143, New York, NY, USA, 2008. ACM.

[64] Y Yu and J. Kienzle. Towards an efficient aspect precedence model. In *Proceeding of the dynamic aspects workshop*, pages 156–167, 2004.