ASPECTMATLAB: AN ASPECT-ORIENTED SCIENTIFIC PROGRAMMING LANGUAGE

by

Toheed Aslam

School of Computer Science
McGill University, Montréal

February 2010

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2010 Toheed Aslam

Abstract

There has been relatively little work done in the compiler research community for incorporating aspect-oriented features in scientific and dynamic programming languages. MATLAB® is a dynamic scientific programming language that is commonly used by scientists because of its convenient and high-level syntax for arrays, the fact that type declarations are not required, and the availability of a rich set of application libraries. This thesis introduces a new aspect-oriented scientific language, AspectMatlab.

AspectMatlab introduces key aspect-oriented features in a way that is both accessible to scientists and where the aspect-oriented features concentrate on array accesses and loops, the core computation elements in scientific programs. One of the main contributions of this thesis is to provide a compiler implementation of the AspectMatlab language. It is supported by a collection of scientific use cases, which demonstrate the potential of aspect-orientation for scientific problems.

Introducing aspects into a dynamic language such as MATLAB also provides some new challenges. In particular, it is difficult to statically determine precisely where patterns match, resulting in many dynamic checks in the woven code. The AspectMatlab compiler uses flow analyses to eliminate many of those dynamic checks.

This thesis reports on the language design of AspectMatlab, the *amc* compiler implementation, and also provides an overview of the use cases that are specific to scientific programming. By providing clear extensions to an already popular language, AspectMatlab will make aspect-oriented programming accessible to a new group of programmers including scientists and engineers.

Résumé

Relativement peu de travail a été accomplis dans le milieu de la recherche du compilateur pour l'intégration des caractéristiques orientées à l'aspect dans les domaines scientifique et dynamique des langages de programmation. MATLAB® est un langage de programmation scientifique dynamique qui est couramment utilisé par les scientifiques en raison de sa pratique et la syntaxe de qualité pour des tableaux ; du fait que les déclarations de type ne sont pas nécessaires, et de la disponibilité de vastes bibliothèques d'applications. Cette thèse introduit un nouvel aspect de langue de recherche scientifique : AspectMatlab.

AspectMatlab introduit fonctionnalités d'aspect orientées d'une manière qui est à la fois accessible aux scientifiques et où les fonctionnalités d'aspect orientées se concentrent sur les accès réseau et des boucles, les éléments de calcul de base dans les programmes scientifiques. L'une des principales contributions de cette thèse est de fournir une implémentation du compilateur du langage AspectMatlab. Il est soutenu d'une collection de cas d'utilisation scientifique, qui montre le potentiel de l'orientation aspect pour des problèmes scientifiques.

L'introduction des aspects dans un langage dynamique comme MATLAB représente aussi quelques nouveaux défis. En particulier, il est difficile de déterminer statiquement où les modèles concident, résultant dans de nombreux contrôles dynamiques dans le code tissé. Le compilateur d'AspectMatlab utilise le flux d'analyses pour éliminer un grand nombre de ces contrôles dynamiques.

Cette thèse signale la conception du language d'AspectMatlab et l'implementation du compilateur *amc*. Elle fournit également un aperu de l'utilisation des cas qui sont spécifiques à la programmation scientifique. En fournissant des extensions claires avec un langage déjà

populaire, AspectMatlab rendra la programmation orientée à l'aspect accessible à un nouveau groupe de programmeurs y compris des scientifiques et des ingénieurs.

Acknowledgements

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC).

We acknowledge contributions by M.Sc. student Jesse Doherty, implementor of the name resolution analysis and M.Sc. student Anton Dubrau, for the example scientific use cases.

We acknowledge the support and guidance given by our thesis supervisor, Professor Laurie Hendren.

Table of Contents

Abstract **i**

Résumé **iii**

Acknowledgements **v**

Table of Contents **vii**

List of Figures **xi**

List of Tables **xiii**

Table of Contents **xv**

1 Introduction **1**

1.1 Contributions 3

1.2 Thesis Outline 3

2 Language Definition **5**

2.1 Aspects 6

2.2 Patterns 9

2.2.1 Function Patterns . . . 11
2.2.2 Array Patterns . . . 12
2.2.3 Selective Matching . . . 14
2.2.4 Loop Patterns . . . 15
2.2.5 Scope Patterns . . . 17
2.2.6 Compound Patterns . . . 18
2.3 Actions . . . 19
2.3.1 Context Exposure . . . 20
2.3.2 `Around` Actions . . . 21
2.3.3 Precedence Order . . . 23
2.4 Small Example . . . 24

3 Scientific Use Cases **27**
3.1 Tracking operations that grow arrays . . . 28
3.2 Tracking array sparsity . . . 29
3.3 Measuring floating point operations . . . 34
3.4 Adding units to computations . . . 36
3.5 Interpreting loop iteration space . . . 40
3.6 Other possibilities . . . 42

4 Compiler **45**
4.1 Compiler Structure . . . 45
4.2 Separator & AspectInfo . . . 47
4.3 Transformations . . . 48
4.3.1 Expression Simplification . . . 48

4.3.2 Loop Rewriting . . . 49
4.4 Name Resolution Analysis . . . 50
4.5 Matching and Weaving . . . 53
4.5.1 Weaving at the function level . . . 53
4.5.2 Weaving at the loop level . . . 54
4.5.3 Weaving at the statement level . . . 55
4.5.4 Weaving `around` actions . . . 55
4.6 Post-processing . . . 58
4.7 Woven Example . . . 59
4.8 Performance Overhead . . . 62

5 Related Work **65**
5.1 AspectJ . . . 65
5.1.1 Extension: Array specific pointcuts . . . 66
5.1.2 Extension: Multi-dimensional array specific pointcuts . . . 66
5.1.3 Extension: Loop specific pointcuts . . . 67
5.2 AspectCobol . . . 69
5.3 AOP in MATLAB . . . 69
5.4 Summary . . . 70

6 Conclusions and Future Work **73**
6.1 Conclusions . . . 73
6.2 Future Work . . . 75

A AspectMatlab Grammar **77**

B User Manual **81**

B.1 Flags 81

C Directory Structure **83**

D Scientific Aspects **87**

D.1 Tracking operations that grow arrays 87

D.2 Tracking array sparsity 91

D.3 Measuring floating point operations 96

D.4 Adding units to computations 102

D.5 Interpreting loop iteration space 110

Bibliography **113**

List of Figures

2.1 Syntax of an Aspect 8

2.2 Syntax of Patterns 10

2.3 Function Join Points 12

2.4 Array Join Points 13

2.5 Array Join Points - Order 13

2.6 Loop Join Points 16

2.7 Syntax of Actions 19

2.8 Actions Precedence Order 24

2.9 Aspect to count all calls made with at least 2 arguments 25

2.10 Simple MATLAB Function 26

3.1 Outline of array growing aspect 30

3.2 Output of the array growing benchmark 31

3.3 Outline of sparsity aspect 32

3.4 Output of the sparsity benchmark 33

3.5 Outline of flops aspect 35

3.6 Output of the flops benchmark 37

3.7 Outline of units aspect 38

3.8 Example of units aspect 39

3.9 Outline of loops aspect 41

3.10 Example of loops aspect 42

3.11 Output of loops aspect 42

4.1 Overall structure of the *amc* AspectMatlab compiler 46

4.2 Aspect with multiple patterns on the same entity 51

4.3 Weaving without Name Resolution Analysis 51

4.4 Weaving with Name Resolution Analysis 52

4.5 Matching and Weaving process outline 53

4.6 Example of an **around** function 56

4.7 Aspect for multiple **around** actions 57

4.8 Translated multiple **around** functions 58

4.9 MATLAB class generated from the aspect 60

4.10 Woven MATLAB function 61

5.1 Example of AspectJ multi-dimensional array pointcuts (from [CC07]) . . . 67

5.2 Example of AspectJ loop pointcut (from [HG06]) 68

5.3 Example of AspectCobol (from [LDS05]) 70

List of Tables

2.1 List of Primitive Patterns 11

2.2 Selective Pattern Matching 15

2.3 Context Selectors with respect to Join Points 21

4.1 Performance overhead 63

List of Listings

2.1 A typical MATLAB class example 6

2.2 A typical aspect example 8

2.3 Function Patterns 12

2.4 Array Patterns 14

2.5 Selective Matching 14

2.6 Loop Patterns 17

2.7 Example of loop patterns 17

2.8 Scope Patterns 18

2.9 Compound Patterns 18

2.10 Before and After Actions 20

2.11 Context Exposure 20

2.12 An around action without `proceed` 22

2.13 An around action with `proceed` 22

Chapter 1

Introduction

MATLAB® is a programming language that provides scientists with an interactive development loop, high-level array operations and a rich collection of built-in and library functions [Mat]. MATLAB is also a very dynamic language in which variable types are not declared, and in which new functions and scripts are loaded dynamically. Although MATLAB recently incorporated object-oriented programming features, there are currently no aspect-oriented features.

Our challenge was to define and implement a new aspect-oriented programming language that was a natural extension of MATLAB. We wanted to build upon the successes of languages such as AspectJ[Asp03, KHH+01], but at the same time tailor our approach to the needs of the scientific programmer. In particular, we wanted to introduce new language features for matching array and loop operations, both of which are central to scientific programming. We also wanted to introduce aspect-oriented programming in a way that was a natural extension to the MATLAB language and so that it would be understood and adopted by the scientific programmers.

AspectMatlab is a component of a larger effort known as the McLab project[1]. The overall goal of the project is to find ways to improve the performance, usefulness and accessibility of current scientific programming languages.

[1] www.sable.mcgill.ca/mclab

We have defined an extension of MATLAB, AspectMatlab, which supports the notions of *patterns* (pointcuts in AspectJ terminology), and *named actions* (advice in AspectJ terminology). An aspect in AspectMatlab looks very much like a class in the object-oriented part of MATLAB. Just like classes, an aspect can have properties (fields) and methods. However, in addition, the programmer can specify patterns (pointcuts) and before, after and around actions (advice). Each action is declared with a name (unlike advice in AspectJ, which do not have names).

AspectMatlab supports traditional patterns (pointcuts) such as `call` and `execution`, but we have also concentrated on an effective design for `get` and `set` patterns which naturally deal with arrays. Loops are key control structures in scientific programs and we have developed a collection of patterns which allow one to match on loops in a variety of ways. We have also been inspired by AspectCobol [LDS05] in that we expose join point context information via selectors that are associated with actions.

In order to motivate our new patterns, we have developed a collection of use cases which we believe illustrates uses that are specific to scientific programming.

We have implemented the *amc* compiler which translates AspectMatlab source files to pure MATLAB source files. The generated code can be run using any MATLAB system. The overall structure of the compiler was inspired from the abc [ACH+05, abc] system and is built as an extension of the McLab MATLAB front-end. In implementing the compiler it became clear to us that weaving into MATLAB code offers new challenges that are different from weaving into more statically-typed, traditional languages such as Java. As one example, the expression `a(i)` may be either a call to function `a` or a get of the `i`'th element of array `a`. Even worse, the precise rules for looking up names differs for functions, inner functions and scripts. Thus, a naive weaving strategy for MATLAB requires a lot of dynamic checks to determine if an expression matches.

To deal with the special challenges of weaving in MATLAB, we have utilized some intra-procedural flow analyses using the McLab analysis framework, developed at Sable Research Lab, which enables us to statically determine whether names correspond to variables or functions. Applying these analyses before weaving allows us to greatly reduce the

number of dynamic checks required.

1.1 Contributions

This thesis makes the following contributions:

- Design of an aspect-oriented extension to a scientific programming language, MATLAB.
- Design and implementation of an extensible AspectMatlab Compiler, *amc*, enriched with a set of aspect-oriented features.
- Introduction of new scientific patterns to cross-cut the concerns related to arrays and loops. Both of these constructs are essential parts of a scientific language.

Finally, we aim to make *amc* a viable aspect-oriented compiler which should become increasingly usable by end-users for real-world scientific applications. Based on our experience with AspectMatlab, we propose promising future directions for dynamic languages to adopt aspect-oriented features. We identify key factors in our implementation and propose ways to improve upon the performance results we have obtained with the AspectMatlab Compiler.

1.2 Thesis Outline

This thesis is divided into 6 chapters (including this introduction chapter). *Chapter 2* introduces the AspectMatlab language and discusses key structures of the language in detail. In *Chapter 3* we present some use-cases to demonstrate the importance of an aspect-oriented language for a scientific programming language. *Chapter 4* examines the AspectMatlab compiler's architecture and its different phases in detail. It also discusses issues associated with the MATLAB programming language design that make matching and weaving difficult. Chapter 5 discusses related work done in some other languages, which helped us to

form the base of our research, and the ways in which our approach differs with them. Finally, chapter 6 presents our conclusions and outlines some possible future research work in this domain beyond what we have achieved.

Chapter 2

Language Definition

Although AspectMatlab's design is mostly inspired by AspectJ, there are distinctive features of our language which are based upon two driving principles: (1) the ability to cross-cut multidimensional MATLAB array accesses and loops, and (2) the ability to bind context information from the join point shadow as part of the action declaration. While designing the syntax for the aspect constructs, we focused on achieving a couple of goals. First, enriching the patterns structure for enhanced selective matching and secondly, not to deviate from the existing language constructs for the sake of better accessibility for existing MATLAB programmers.

This chapter elaborates the design of an aspect-oriented extension to a scientific programming language. We discuss the structure of an Aspect and all the constructs an aspect may contain, i.e., Properties, Patterns, Methods and Actions. It provides important details about the features of MATLAB we extended and on which we based our design. We discuss supported types of patterns and actions in detail. We also describe ways to create compound user-defined patterns and how to weave actions in different orders.

2.1 Aspects

In AspectMatlab, aspects are defined using a syntax similar to MATLAB classes. A MATLAB class typically contains properties, methods and events. As in other object-oriented languages, properties in MATLAB class encapsulate the data that belongs to instances of classes, which can be assigned default values, initialized in class constructors, and used throughout the class. Data contained in properties can be declared public, protected, or private. This data can be a fixed set of constant values, or it can be dependent on other values and calculated only when queried. Different attributes can be applied over a block of properties and property-specific access methods can be specified.

Encapsulation using methods is also a familiar concept in an object-oriented systems. MATLAB class methods are a little different as they act as an enclosing block, which can host a variety of functions. Common types of methods are ordinary functions, class constructors, class destructors and property access functions. Method blocks can be configured with different attributes, including access specifiers.

Listing 2.1 shows a typical MATLAB class, `myClass`, which can be used as a simple counter. This class declares a property, `count`, which has default scalar value 0. The counting functionality is provided through two functions. `incCount` increments the counter and `getCount` can be used to query the current value of count, which is returned through variable `out`. One important point to notice here is that MATLAB class methods always have the calling object automatically passed as the first argument.

```
1  classdef myClass
2
3  properties
4    count = 0;
5  end
6
7  methods
8
9    function incCount(this)
10     this.count = this.count + 1;
```

```
11   end
12
13   function out = getCount(this)
14     out = this.count;
15   end
16
17 end %methods
18
19 end %classdef
```

Listing 2.1 A typical MATLAB class example

In this chapter, we outline the grammar of AspectMatlab in pieces as we go through related concepts and constructs. If you have a coloured version of this document, you will see that all references to productions in the McLab implementaion of the base MATLAB grammar, are given in red. The complete grammar specification is provided in Appendix A.

As shown in *Figure 2.1*, the base McLab `program` rule is extended to include aspects, along with functions, scripts and classes, as a program entity. Just like a MATLAB class structure, an `aspect` is named and contains a body. An aspect retains the properties and methods constructs, while adding two aspect-related constructs: patterns and actions. Patterns are formally known as pointcuts in AspectJ and are used as picking out certain join points in the program flow. AspectMatlab actions correspond to AspectJ advice, which essentially is a block of code intended to be executed at certain points in the program. This choice of terminology was intended to convey that patterns specify where to match and actions specify what to do.

Moreover, it is important to explain the `stmt_separator` non-terminal, imported from McLab. Unlike other high level languages, a MATLAB statement can be terminated in multiple ways. These statement separators include the new-line, a comma or a semi-colon.

With the addition of patterns and actions, Listing 2.2 shows an extension to the class presented in Listing 2.1. `myAspect` counts how many times a function named `foo` is invoked. To achieve this functionality, we first define a pattern. Pattern `callFoo` provides us the way to specify the target join points. Once we match such join points in the source

⟨program⟩ ::⇒⟨script⟩ | ⟨function_list⟩ | ⟨class⟩ | ⟨aspect⟩
⟨aspect⟩ ::⇒'aspect' IDENTIFIER ⟨stmt_separator⟩ ⟨help_comment⟩
    ⟨aspect_body⟩* 'end'
⟨aspect_body⟩ ::⇒
    ⟨properties_block⟩ ⟨stmt_separator⟩
    | ⟨patterns_block⟩ ⟨stmt_separator⟩
    | ⟨methods_block⟩ ⟨stmt_separator⟩
    | ⟨actions_block⟩ ⟨stmt_separator⟩

Figure 2.1 Syntax of an Aspect

code, then we can call the corresponding action, `actCall`. This action triggers before the call to function `foo` and increments the counter.

```
aspect myAspect

properties
  count = 0;
end

patterns
  callFoo : call(foo);
end

methods
  function incCount(this)
    this.count = this.count + 1;
  end
end

actions
  actCall : before callFoo
    this.incCount();
  end
end

end
```

Listing 2.2 A typical aspect example

In the compiled code, an aspect is transformed into a class and the actions are translated into corresponding methods of the resulting class. As described earlier, MATLAB class

methods have the invoking class object as an argument. So the methods created out of actions are also provided that object, which we named `this` for the purposes of clarity and consistency. Inside an action body, `this` should be used to interact with the properties and methods for the specific object.

We present a detailed discussion on patterns and actions in the following sections.

2.2 Patterns

Just like any other aspect-oriented language, AspectMatlab provides a variety of patterns that can be used to match basic language constructs. In addition to standard patterns such as those supported by AspectJ, a scientific programming language like MATLAB possesses other important cross-cutting concerns. In MATLAB, array constructs are heavily used and programs are written in the form of large functions or scripts containing many loops.

Grammar rules for patterns are presented in *Figure 2.2*. Patterns are contained inside blocks, and an aspect can contain any number of such blocks of patterns. A pattern is formed by its unique name and the pattern designators. AspectMatlab provides a number of primitive patterns targeting different constructs of MATLAB. These primitive patterns can be logically combined to form the compound pattern designators. We will discuss this concept in detail in *Section 2.2.6*.

While providing the basic function-related patterns like **`call`** and **`execution`**, we also introduce two new sets of patterns: (1) **`get`**/**`set`** patterns, enabling the facility to capture array-related operations along with useful context exposure; and (2) loop patterns, which will help programmers to handle the loop iteration space and details of loop-intensive computation. AspectMatlab also supports a special **`within`** pattern, which allows us to restrict the scope of matching to certain constructs of the source code, such as functions, scripts, classes or loops.

Towards the bottom of *Figure 2.2*, we introduce some grammar rules to enable a programmar to perform selective matching. MATLAB syntax allows us to make a function call, without even providing the exact number of parameters specified in the function signature.

⟨patterns_block⟩ ::⇒'patterns' ⟨stmt_separator⟩ ⟨patterns_body⟩* 'end'
⟨patterns_body⟩ ::⇒IDENTIFIER ':' ⟨pattern_designators⟩ ⟨stmt_separator⟩
⟨pattern_designators⟩ ::⇒
    ⟨pattern_designators_and⟩
    | ⟨pattern_designators⟩ '|' ⟨pattern_designators_and⟩
⟨pattern_designators_and⟩ ::⇒
    ⟨pattern_designators_unary⟩
    | ⟨pattern_designators_and⟩ '&' ⟨pattern_designators_unary⟩
⟨pattern_designators_unary⟩ ::⇒
    ⟨pattern_designator⟩
    | '∼' ⟨pattern_designator⟩
⟨pattern_designator⟩ ::⇒
    '(' ⟨pattern_designators⟩ ')'
    | 'set' '(' ⟨pattern_select⟩ ')'
    | 'get' '(' ⟨pattern_select⟩ ')'
    | 'call' '(' ⟨pattern_select⟩ ')'
    | 'execution' '(' ⟨pattern_select⟩ ')'
    | 'mainexecution' '(' ')'
    | 'loop' '(' ⟨pattern_select⟩ ')'
    | 'loopbody' '(' ⟨pattern_select⟩ ')'
    | 'loophead' '(' ⟨pattern_select⟩ ')'
    | 'within' '(' ⟨construct_type⟩ ','⟨pattern_select⟩ ')'
    | IDENTIFIER
⟨pattern_select⟩ ::⇒
    ⟨pattern_target⟩
    | ⟨pattern_target⟩ '(' ⟨list_dotdot⟩ ')'
⟨pattern_target⟩ ::⇒
    ⟨pattern_target_unit⟩
    | ⟨pattern_target⟩ ⟨pattern_target_unit⟩
⟨pattern_target_unit⟩ ::⇒'*' | IDENTIFIER
⟨list_dotdot⟩ ::⇒ε
    | '..'
    | ⟨list_star⟩
    | ⟨list_star⟩ ',' '..'
⟨list_star⟩ ::⇒'*'
    | ⟨list_star⟩ ',' '*'
⟨construct_type⟩ ::⇒'*' | 'function' | 'script' | 'loops'
    | 'class' | 'aspect'

Figure 2.2 Syntax of Patterns

Arrays can be indexed in a similar fashion. So AspectMatlab provides a functionality to enhance the matching based on the actual parameters/indices involved. We explain the idea of selective matching in *Section 2.2.3*.

Moreover, matching can be performed based on the expressions containing the wild card symbol "*", which results in a broader scope of matching.

A list of primitive patterns supported by AspectMatlab is presented in *Table 2.1*. We discuss the different kinds of patterns in the following sections.

functions	call execution mainexecution	captures calls to functions or scripts captures the execution of function or script bodies captures the execution of the main function or script body
arrays	get set	captures array accesses captures array assignments
loops	loop loophead loopbody	captures execution of a whole loops captures the header of a loop captures the body of a loop
scope	within	restricts the scope of matching

Table 2.1 List of Primitive Patterns

2.2.1 Function Patterns

AspectJ and other aspect-oriented languages provide basic function-related cross-cutting features, which enable a programmer to track, for example, the calls made to all or some specific functions matching the specified pattern. Other places of interest in a function source code are the entry and exit points of the body.

Figure 2.3 shows an example of the function-related join points in the source code. The whole body of the function `main` matches an **`execution`** pattern, whereas every call to a function is captured by the **`call`** pattern.

AspectMatlab also supports both **`call`** and **`execution`** patterns, not only for functions but to cross-cut scripts as well. Because there is no specific main entry point to MATLAB

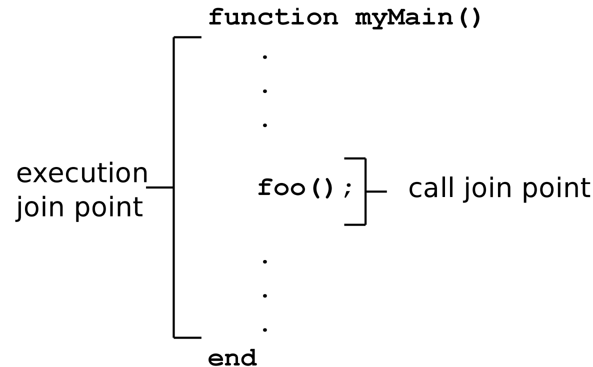


Figure 2.3 Function Join Points

programs, so we introduce a `mainexecution` pattern. This pattern will match the execution of the main function or script, (i.e., the first function or script executed). The function patterns given in Listing 2.3 show example uses, where `pCallFoo` pattern matches all calls made to the function or script named `foo` and `pExecutionMain` pattern captures the entry and exit points of the main function.

```
1 patterns
2   pCallFoo : call(foo);
3   pExecutionMain : mainexecution();
4 end
```

Listing 2.3 Function Patterns

2.2.2 Array Patterns

AspectJ provides array pointcuts functionality. However, the pointcuts of AspectJ do not support array objects in full. When an element of an array object is set or referenced, the corresponding index values and the assigned value are not exposed to the advice. AspectJ was extended to add array pointcuts but these extensions either just work for one-dimensional arrays or they force programmers to use other pointcuts in order to be able to perform selective matching and to fetch context information.

In contrast, AspectMatlab provides simple, yet powerful, patterns to capture array accesses, **get** and **set**. As shown is *Figure 2.4*, the first assignment statement is a **set** join point

where `arr1` is being assigned a new value. The second statement is also a **set** join point for `arr2`, but the right hand side actually reads `arr1`. So the right hand side of the second assignment statement is a **get** join point.

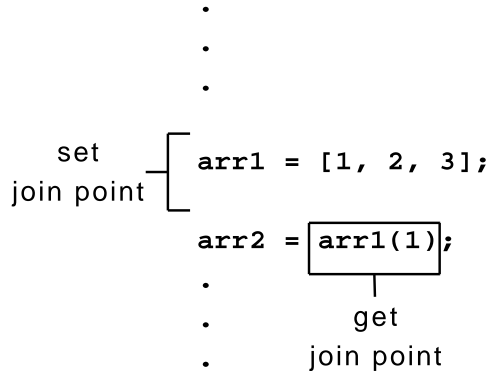


Figure 2.4 Array Join Points

Figure 2.5 shows an example of a more complicated **get** match. Here we have array accesses within another array access and we have to sort out the order in which all these join points are matched. We decided to follow the evaluation order of an expression, where all the sub-expressions are evaluated before the containing expression. So, the first **get** join point in the second assigment statement is the access of `x`, followed by the second **get** join point for `y` and finally, the third **get** join point is the whole right hand side.

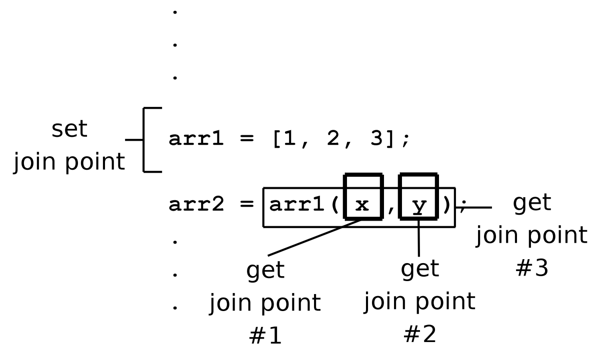


Figure 2.5 Array Join Points - Order

Examples of array patterns are given in Listing 2.4. Pattern `pGetX` matches all the join

points in the source code where any array or MATLAB matrix access operation is performed. Similarly, all the write operations on the arrays can be captured using pattern pSetX.

```
1 patterns
2   pGetX : get(*);
3   pSetX : set(*);
4 end
```

Listing 2.4 Array Patterns

2.2.3 Selective Matching

As compared to other aspect-oriented languages, AspectMatlab eliminates the need of a separate pattern for capturing arrays and then using another pattern to specialize the matching. In MATLAB a function call does not necessarily have to provide exactly as many arguments as specified in a function signature. Also in the case of array operations, sub-arrays can be accessed by providing fewer dimensions than the actual dimensions of an array.

Moreover, the syntax to make a function/script call and array access in MATLAB is the same. So the pattern specification grammar was enriched to incorporate matching based upon the number of arguments involved. *Section 2.2.1* and *Section 2.2.2* describe simple function and array-related patterns. In this section, we provide examples of more selective matching.

As shown in the Listing 2.5, pattern call2args will match all calls, but only the ones made with two or more parameters, thus ignoring the calls with one or no parameters. If we want to match on all the arrays which are being initialized or replaced completely, pattern fullSet will help us achieve that.

```
1 patterns
2   call2args : call(*(*,..));
3   fullSet : set(*());
4 end
```

Listing 2.5 Selective Matching

AspectJ also provides this facility of selective matching, but it uses separate notations for different pointcuts. The MATLAB syntax allows us to come up with a general matching notation applicable for both call/execution and get/set patterns. A list of possible use cases of such matching for the call pattern is given in *Table 2.2*.

call(foo)	matches all calls to `foo` (function or script)
call(foo())	matches calls with no arguments (function or script)
call(foo(*))	matches calls with exactly one argument (function only)
call(foo(..))	matches calls with 1 or more argument(s) (function only)
call(foo(*,..))	matches calls with 2 or more arguments (function only)
	...and so on
set(arr)	matches all assignments to `arr`
set(arr())	matches assignments with no indices
set(arr(*))	matches assignments with exactly one index
set(arr(..))	matches assignments with 1 or more index/indices
set(arr(*,..))	matches assignments with 2 or more indices
	...and so on

Table 2.2 Selective Pattern Matching

2.2.4 Loop Patterns

The original AspectJ language definition did not contain any loop-related pointcuts. In MATLAB, loops are extensively used and having the ability to cross-cut the loops is equally important in such a language. AspectMatlab provides a range of poincuts for loops: `loop`, `loopbody` and `loophead`.

As shown in *Figure 2.6*, the `loop` join point presents only an outside view of the loop; because the points before and after the loop are not within the loop itself. For some applications it might be desirable to advise the loop body. Also, the loop iterators are good candidates to be advised. Because in MATLAB, loop headers are evaluated completely before the loop itself. So the `loophead` join point is not contained inside the `loop` join point.

In aspect-oriented systems, the means of selection for a join point is, in most cases, ulti-

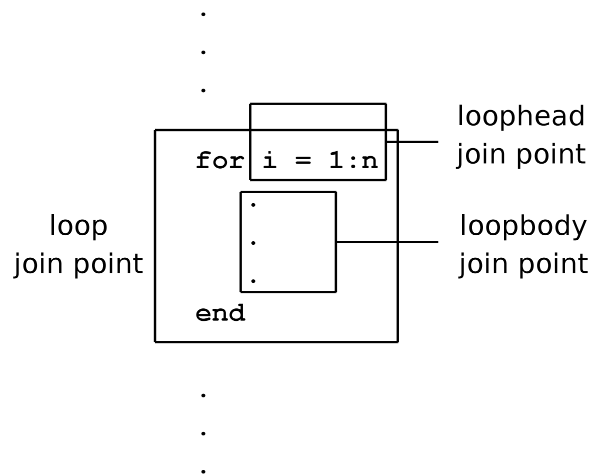


Figure 2.6 Loop Join Points

mately based on the naming of some source element characterising the join point, possibly using a regular expression. For example, to advise a method call or a group of methods, the pointcut has to contain an explicit reference to some names characterising the method signatures, for instance, a pattern matching the name of the methods. Since loops can not be named in MATLAB, a name-based pattern to write a pointcut that would select a particular loop will not work.

If it is known for certain that all the loops within a function are to be advised, it would be possible in AspectMatlab to use certain scope-related pattern to restrict the loop pattern to all the loops contained in the functions picked up in the restricted scope. However, selecting only one of several loops within the same function turns out to be impossible without any further mechanism. So for the sake of loops identification, we decided to use the loop iterator variables to match a loop pattern.

Examples of simple loop patterns are given in Listing 2.6. All three patterns will match on all the loops, either `for` or `while`, which iterate on variable `i`.

```
patterns
  pLoopI : loop(i);
  pLoopHeadI : loophead(i);
  pLoopBodyI : loopbody(i);
end
```

Listing 2.6 Loop Patterns

For example, consider the two loops shown in Listing 2.7, where both display the numbers from 1 to 10. Both loops match the patterns given in Listing 2.6.

```
for i = 1:10
  disp(i);
end

i=1;
while (i<=10)
  disp(i);
  i = i+1;
end
```

Listing 2.7 Example of loop patterns

2.2.5 Scope Patterns

There are certain cases in aspect-oriented systems, where some built-in language features are required to restrict the scope of matching of the patterns. For example, in AspectMatlab we use loop iterator variables to identify loops. The question might arise that names for loops iterator variables are often very general (for example, i or j), so we might end up over-matching loops unintentionally. The `within` pattern comes in very handy in such situations to restrict the scope of matching to specific constructs.

AspectMatlab supports a list of MATLAB constructs, such as `function`, `script`, `class`, `aspect` and `loops`.

Listing 2.8 presents examples of different cases of the **`within`** pattern. The `pWithinFoo` pattern will match every kind of join point, only inside the function `foo`. Similarly, the `pWithinBar` pattern will match every join point inside the script `bar` and the `pWithinMyClass` pattern will match every join point inside the class `myClass`. The `pWithinLoops` pat-

tern captures all join points within all the loops. Lastly, pWithinAllAbc will restrict the scope to all kinds of constructs, which are named abc.

```
1 patterns
2   pWithinFoo : within(function, foo);
3   pWithinBar : within(script, bar);
4   pWithinMyClass : within(class, myClass);
5   pWithinLoops : within(loops, *);
6   pWithinAllAbc : within(*, abc);
7 end
```

Listing 2.8 Scope Patterns

2.2.6 Compound Patterns

As in other aspect-oriented languages, AspectMatlab also provides a programmer the facility of creating compound patterns. Such user-defined patterns are in fact logical combination of user-defined patterns and primitive patterns given in *Table 2.1*.

Examples of compound patterns given in Listing 2.9 display the level of flexibility a programmer can achieve in order to create different logical compounds of primitive patterns. Pattern pCallFoo matched all calls made to function foo, but only the ones from within the loops, either for or while loops. On the other hand, the pattern pGetOrSet will match all array read or write operations, but the ones only within the function bar. pCallExec shows a combination of an already defined pattern pCallFoo with a primitive pattern **execution**.

```
1 patterns
2   pCallFoo : call(foo) & within(loops, *);
3   pGetOrSet : (get(*) | set(*)) & within(function, bar);
4   pCallExec : pCallFoo | execution(foo);
5 end
```

Listing 2.9 Compound Patterns

Care should be taken while ANDing patterns of different kinds, because a shadow in the source code has only one specific type. For example, replacing OR with an AND in the pattern pGetOrSet above will result in no match, simply because an array can either be read or written to, not both at the same time.

2.3 Actions

An action is simply a named piece of code which is executed at certain points in the source code, matched by the specified patterns. An aspect can contain many actions, and as in other aspect-oriented languages, there are `before`, `around` and `after` actions.

As shown in *Figure 2.7*, an aspect can contain any number of action blocks, which in turn can host multiple actions inside them. Unlike AspectJ, actions in AspectMatlab are named. Besides the name, an action is linked to a named pattern defined in the patterns block. The type of an action specifies the weaving point of an action with respect to the join points against the pattern specified. Just like a regular MATLAB function, an action can have input parameters. These parameters are special context information which is fetched from the static shadow of each join point matched. Context exposure is described in detail in *Section 2.3.1*.

⟨actions_block⟩ ::⇒'actions' ⟨stmt_separator⟩ ⟨actions_body⟩* 'end'
⟨actions_body⟩ ::⇒
IDENTIFIER ':' ⟨action_type⟩ IDENTIFIER ⟨stmt_separator⟩
⟨help_comment⟩ ⟨stmt_or_function⟩ 'end'
| IDENTIFIER ':' ⟨action_type⟩ IDENTIFIER ':' ⟨input_params⟩
⟨stmt_separator⟩ ⟨help_comment⟩ ⟨stmt_or_function⟩ 'end'
⟨action_type⟩ ::⇒'before' | 'after' | 'around'

Figure 2.7 Syntax of Actions

Simple examples of named `before` and `around` actions, which correspond to the patterns `pCallFoo` and `pExecutionMain` described in *Section 2.2.1*, are given in Listing 2.10. The action `aCountCall` will be weaved in just before each call to function `foo`. This action simply increments the `count` property defined in the properties block of the aspect. Now if we want to display the total number of calls made at the end of the program, we can use the `aExecution` action. Assuming the end of function `main` as the program exit point, `aExecution` action will be weaved in just after the whole function body.

```
actions
  aCountCall : before pCallFoo
    this.count = this.count + 1;
  end

  aExecution : after executionMain
    total = this.getCount();
    disp(['total calls: ', num2str(total)]);
  end
end
```

Listing 2.10 Before and After Actions

2.3.1 Context Exposure

When it comes to capturing the context of a join point, AspectCobol's [LDS05] design doesn't rely on the use of reflection inside the advice code, as performed in AspectJ [KHH+01]. Rather, it suggests that join point reflection on the static shadow should be a part of the pointcut. The extraction of the context-specific information is described as part of the pointcut designator. We extend the idea of binding the results of desired context variables for subsequent use in the action code.

In AspectMatlab, access to the static program context that belongs to the join point is selector based. These selectors are specified along with an action definition, because an action corresponds directly to the static join point shadow. In the example below, the action `actcall`, which acts before the join points matching the pattern `call2args` given in Listing 2.5. It will fetch the `name` and `args` of the function call from the join point shadow.

```
actcall : before call2args : (name, args)
  %
  disp(['calling ', name, ' with arguments(', args , ')']);
  %
end
```

Listing 2.11 Context Exposure

Of course, a selector is only applicable depending upon the join point type. For example, the `counter` selector is only meaningful when used on a loop join point. The `args`

selector fetches the array indices in case of array patterns, whereas the same selector is used to get the function arguments/parameters in case of function patterns.

A list of context selectors and their meaning with different join points is given in *Table 2.3*.

	set	get	call	execution	loop	loopbody	loophead
args	indices		arguments passed		loop iteration space		
obj	variable before set	variable	function handle	-	-	iterator variable	-
newVal	new array	-	-	-	-	-	loop range
counter	-	-	-	-	-	current iteration	-
name	name of the entity matched				-	-	-
pat	name of the pattern matched						
line	line number in the source code						
loc	enclosing function/script name						
file	enclosing file name						
aobj	variable name	-	-	-	-	-	-
ainput	-	input var name(s)			-	-	-
aoutput	-	-	-	output var name(s)	-	-	-
varargout	cell array variable used to return data from `around` action						

Table 2.3 Context Selectors with respect to Join Points

2.3.2 `Around` Actions

Consider the `before` action given in *Section 2.3.1*, which is woven in just before the actual call to any function with 2 or more arguments. What if we want to manipulate the arguments before making such calls, or we want to add more arguments to the call, or we want to provide fewer arguments, or we want to make such a call more than one time, or we want to call some other function instead, or we just don't want to make such function calls?

The `around` actions are the answer to all the questions. An `around` action is executed *instead of* the actual join point matched. All the valid context information can be fetched in the around action and then used accordingly. The actual join point can still be executed from within an around action, using a special `proceed` call. The `proceed` function can be called any number of times or not at all.

The `around` actions can be used with all AspectMatlab supported pattern types, except

some patterns inside the script files due to MATLAB semantics. The `around` actions on such join points require these join points to be moved into a separate function, which is not possible inside a script. Unlike `before` and `after` actions, `around` actions can return data. A special MATLAB variable, `varargout`, is used for this purpose; which allows us to return multiple arguments. The `proceed` takes care of the returning arguments, but `varargout` should be set manually in case there is no `proceed`. `varargout` is a list of output values, so it needs to be made sure that it contains as many values as the original join point would return.

For example, the `around` action given in Listing 2.12 captures all calls to `foo` and instead calls `bar` with the same arguments. A single value returned from `bar` is set in `varargout` variable.

```
1 actions
2   actcall : around callFoo : (args)
3     % proceed not called, so varargout is set
4     varargout{1} = bar(args{1}, args{2});
5   end
6 end
```

Listing 2.12 An around action without `proceed`

Listing 2.13 shows the `around` version of the action `actcall` given in Listing 2.11. It simply prints out the function being called along with the arguments, before calling the `proceed`.

```
1 actions
2   actcall : around call2args : (name, args)
3     disp(['before call of ', name, 'with parameters(', args , ')']);
4     proceed();
5     disp(['after call of ', name, 'with parameters(', args , ')']);
6   end
7 end
```

Listing 2.13 An around action with `proceed`

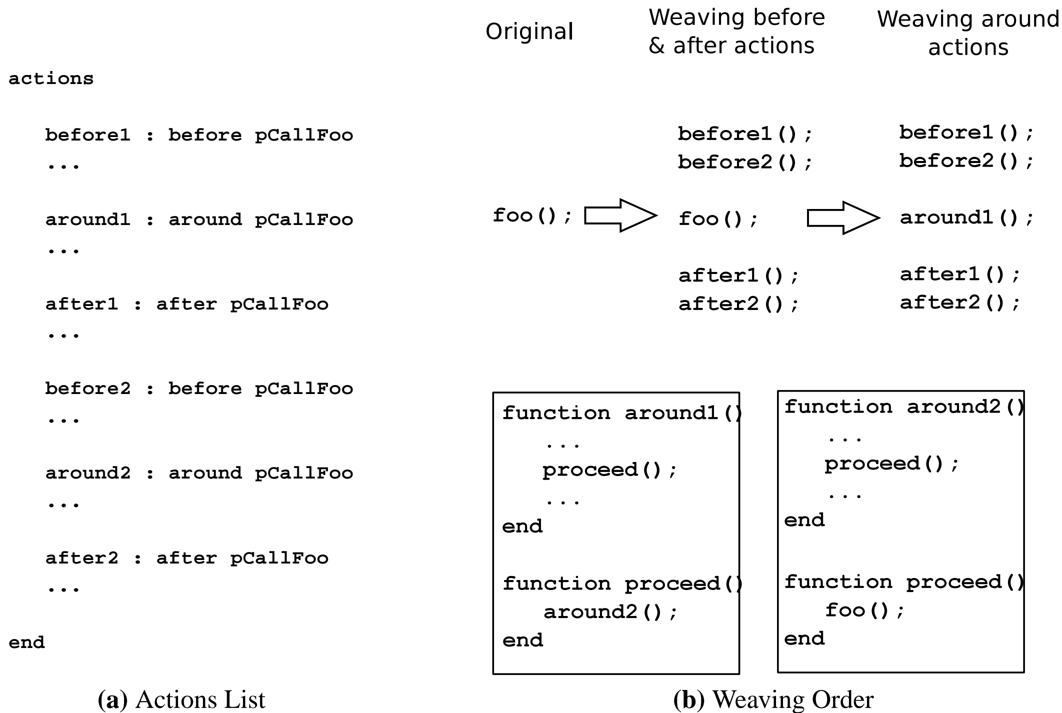
2.3.3 Precedence Order

As shown in *Figure 2.8*, since multiple actions can be triggered at the same join point and if more than one such actions are of the same type, we need default precedence rules for the actions:

- **before** actions are woven just before the join point. In case of multiple **before** actions, the order of the woven advice follows the exact order in which the actions were defined in source code.
- Next, **after** actions are woven just after the join point. In the case of multiple **after** actions, the order of the woven advice follows the exact order in which the actions were defined in the source code.
- Last, multiple **around** actions are woven around the join point in the exact order in which actions are defined in source code. So the outer-most of the **around** actions will be the one appearing first in the woven code and it will go around the next **around** action encountered, or the actual join point if there are no more **around** actions.

In *Figure 2.8a*, multiple actions are targeting a single **call** join point. The weaving points for a join point in the source code are shown in *Figure 2.8b*. All the **before** actions are woven just before the join point in order they are specified. All the **after** actions are woven just after the join point. The call to `foo` is replaced by the call to the first **around** action, which in turn can call the second **around** action through its **proceed** function, and so on.

An important point to notice here is that the default ordering rules of AspectMatlab are simpler and more restrictive than the precedence rules of AspectJ [KHH+01]. However, our action weaving strategy avoids complicated dependency rules, will not lead to any dependency cycles between actions, and is easy to comprehend from a scientific programmer's point of view. Since our actions have names, it would also be simple for us to introduce a declaration to over-ride the default ordering within each of the around, before and after groups.



(a) Actions List

(b) Weaving Order

Figure 2.8 Actions Precedence Order

2.4 Small Example

In *Figure 2.9*, we present an example of an aspect, which counts all the function calls made with at least two arguments. To do so, we need to have a `call` pattern to capture all such calls. The `mainexecution` pattern is used to display the number of calls made at the end of the program.

To demonstrate the application of the aspect from *Figure 2.9*, consider a small base program consisting of the simple MATLAB function given in *Figure 2.10*.

The function `histo` takes one input argument `n` and returns three values `m,s,d`. Values are returned by declaring variables to be return parameters in the function header, then assigning these variables a value. This function first generates some random-sized vectors, then calls several MATLAB functions to generate a histogram, and finally computes some basic statistics.

```
1  aspect myAspect
2
3  properties
4    count=0;
5  end
6
7  methods
8    function out = getCount(this)
9      out = this.count;
10   end
11   function incCount(this)
12     this.count = this.count + 1;
13   end
14 end %methods
15
16 patterns
17   call2args : call(*(*,..));
18   executionMain : mainexecution();
19 end
20
21 actions
22   actcall : around call2args : (name, args)
23     this.incCount();
24     disp(['calling ', name, 'with parameters(', args , ')']);
25     proceed();
26   end
27   actexecution : after executionMain
28     total = this.getCount();
29     disp(['total calls: ', num2str(total)]);
30   end
31 end %actions
32
33 end %myAspect
```

Figure 2.9 Aspect to count all calls made with at least 2 arguments

Once compiled along with the aspect presented in *Figure 2.9*, pattern `call2args` finds only three matching join points (at lines 5, 6 and 9) where the function calls carry two arguments each. So, corresponding action function calls will be woven only at those program points. Note that the function calls with a single input argument (at lines 11, 12 and 13) do not match. Moreover, the action `actexecution` is an **`after`** action, so it will be woven at the end of the function. The woven code generated by the compiler is shown in *Figure 4.10*. It will be easier to follow the output after we explain how different phases of the

```
1  function [m, s, d] = histo(n)
2   % Generate vectors of random inputs
3   % x1 = Normal distribution N(mean=100,sd=5)
4   % x2 = Uniform distribution U(a=5,b=15)
5   x1 = ( randn(n,1) * 5 ) + 100;
6   x2 = 5 + rand(n,1) * ( 15 - 5 );
7   y = x2.^2 ./ x1;
8   % Create a histogram of the results (50 bins)
9   hist(y,50);
10  % Calculate summary statistics
11  m = mean(y);
12  s = std(y);
13  d = median(y);
14 end
```

Figure 2.10 Simple MATLAB Function

compiler work in *Chapter 4*.

Chapter 3

Scientific Use Cases

Scientific programs rely heavily on arrays (i.e., matrices) and loops when performing computations. One of the main goals of AspectMatlab is to expose these language constructs to aspect-oriented programming in order to make it appropriate for use in the scientific computing domain. In this chapter, we show some non-trivial use cases of some typical MATLAB programs that were extended using AspectMatlab. Thus we want to illustrate both the usefulness of aspects in the numerical computing domain in general and the special patterns in particular.

All examples can be found online on our website[1]. They all include the aspects and the programs that are modified, as well as woven code generated by *amc* (i.e., the compiler is not needed to check the benchmarks). Only outlines of the aspects are shown in this chapter, the complete versions are provided in Appendix D. The example benchmarks given in *Section 3.2*, *Section 3.3* and *Section 3.4* were created by a McLab group member, Anton Dubrau, and were reported in a joint paper [TAH10]. The examples in *Section 3.1* and *Section 3.5* are new applications.

In general, we consider two possible use cases: (1) profiling programs, and (2) annotating data to variables in a running program to extend functionality.

Profiling programs is particularly interesting for scientific programs, which are usually

[1] http://sable.mcgill.ca/mclab/aspectmatlab/examples

computationally intensive. Having knowledge about what exactly is going on during execution can help increase efficiency, as shown in *Section 3.1* and the sparsity benchmark (*Section 3.2*). Some information is hard to get by ”traditional” means, i.e., by extending the program to include profiling code. Adding an aspect represents a much cleaner solution, with the additional advantage that they allow one to profile different programs without any modification. Both the sparsity (*Section 3.2*) and flops(*Section 3.3*) examples show this.

With regards to annotating functionality it is interesting to note that the McLab Project was conceived as a framework not only to allow the addition of analyses and compilation of Matlab into different back-ends. It is also a framework to allow the simple development of language extensions, which is exactly what the *amc* compiler represents (an extension of the base MATLAB compiler). Aspects are a quick way to prototype further possible language extensions without much work, as the units (*Section 3.4*) and loops (*Section 3.5*) benchmarks show.

3.1 Tracking operations that grow arrays

MATLAB semantics force the array to be handled in a different way as compared to other objects. Each time the array size increases, MATLAB has to allocate new space to the array. We present a simple example aspect, that is used to track the growing size of the arrays. The purpose of this application is to monitor all the operations which may potentially alter the shape or the size of an array, and in the end we should be able to point to the operations in the source program at which the arrays attain the biggest size.

This aspect is helpful to be able to declare the arrays with their maximum size once at the beginning of the program. It will reduce the performance overhead due to the array copy operations, each time the size changes. In this context, we shall be observing the MATLAB assignment statements. So we need to track the shape and size changes for all the array variables used in the program. The aspect properties are used to keep the profiling information, such as the maximum size of an array along with its location in the source code.

An outline of the aspect is given in *Figure 3.1*. This aspect contains a simple **set** pattern to catch all the array assignments, and another `mainexecution` pattern to display the profiled results at the end of the execution of the program.

The action `aset` acts just before the array assignment operations, and it acquires the new value being set, i.e., the result of the right hand side of the assignment statement, through the use of the context selector `newVal`. The context selector `obj` fetches the current array object, and is used to compare the both old and new array shapes and sizes. We get the name of the array through `name` and the line number of the assignment operation in the source code using `line` context selectors respectively.

The action `displayResults` simply goes through all the data properties which contain the profiled data and prints out the results at the exit point of the program.

We used this array size tracking aspect on an actual program, which utilizes a RungeKutta4 ODE solver [RLB05] to solve the heat equation in 1D given some initial conditions and time interval. The benchmark uses matrices to discretize the heat function in space. So we can track the size and shape change of the matrices used through out the program.

The output of the array size tracking benchmark is given in *Figure 3.2*. As it can be seen in the size change columns, some of the arrays are not that frequently updated because most of such arrays are the input/output parameters of the functions. But few arrays such as `D`, `W` and `t`, increase their size during the execution of the program. The maximum size attained by all the arrays is shown along with the line number of the source code operation.

3.2 Tracking array sparsity

The sparsity benchmark is an aspect which helps to profile how sparse matrices (arrays) are. The sparsity of a matrix is the number of zero elements compared to the number of non-zero elements. If a matrix is sufficiently sparse, it can be stored as a sparse matrix, which is a special data type supported by MATLAB. It stores only the non-zero elements and their location. All arithmetic is supported both on sparse data types and between a mixture of sparse and dense matrices.

```
aspect arrayGrow
  ...
  patterns
    arraySet : set(*);
    exec : mainexecution();
  end

  actions
    ...
    aset : before arrayset : (newVal,obj,name,line,args)
      % create the new obj by applying newVal
      tmp = obj;
      tmp(args{1:numel(args)}) = newVal;
      newVal = tmp;

      % fetch new size of the array
      newSize = numel(newVal);
      oldSize = this.arraySize{id};
      this.arraySize{id} = newSize;

      % update the number of 'set' operations
      this.arraySet{id} = this.arraySet{id}+1;

      % compare the shape change and previous size
      % profile the latest results
      if (~this.sameShape(newVal,obj))
        % how often the dimensions of the array changed
        this.changeShape{id} = this.changeShape{id}+1;
      end
      if (newSize < oldSize)
        % how often the size decreased
        this.decreaseSize{id} = this.decreaseSize{id}+1;
      end
      if (newSize > oldSize)
        % how often the size increased
        this.increaseSize{id} = this.increaseSize{id}+1;
        this.lineNum{id} = line;
        this.maxSize{id} = newSize;
      end
    end

    displayResults : after exec
    %display the results after the execution
  end
end
```

Figure 3.1 Outline of array growing aspect

```
>> program
tracking the operations that grow arrays in the following program...
computation finished

 'var'      'arraySet'    'size decrease'    'size increase'    'max size'    'line#'
 'a'        [       3]    [             0]    [            1]    [       1]    [    1]
 'steps'    [       1]    [             0]    [            1]    [       1]    [    1]
 'tN'       [       1]    [             0]    [            1]    [       1]    [   17]
 'N'        [       4]    [             0]    [            1]    [       1]    [   18]
 'h'        [       3]    [             0]    [            1]    [       1]    [   19]
 'X'        [       1]    [             0]    [            1]    [     299]    [   20]
 'U0'       [       2]    [             1]    [            1]    [     299]    [   21]
 'D'        [       3]    [             1]    [            2]    [   89401]    [   23]
 'tspan'    [       2]    [             0]    [            1]    [       2]    [   56]
 'alpha'    [       2]    [             0]    [            1]    [     299]    [   56]
 'o'        [       2]    [             0]    [            1]    [       1]    [   56]
 'b'        [       2]    [             0]    [            1]    [       1]    [   69]
 'W'        [    1007]    [             2]    [            3]    [  149799]    [   86]
 't'        [    4002]    [             2]    [            2]    [     501]    [   73]
 'j'        [    1002]    [             0]    [            1]    [       1]    [   75]
 'u'        [    4000]    [             0]    [            1]    [     299]    [   24]
 'y'        [    4000]    [             0]    [            1]    [     299]    [   25]
 'k1'       [    1000]    [             0]    [            1]    [     299]    [   78]
 'k2'       [    1000]    [             0]    [            1]    [     299]    [   79]
 'k3'       [    1000]    [             0]    [            1]    [     299]    [   80]
 'k4'       [    1000]    [             0]    [            1]    [     299]    [   81]
```

Figure 3.2 Output of the array growing benchmark

If a matrix is very sparse, then matrix multiplication becomes much cheaper to perform. Since this is where most of the computation of many scientific programs happens, one can achieve order of magnitude speedups in specific instances. Other operations on sparse matrices, like indexing or adding new elements that were previously zero, are much more expensive. This is because they require to traverse or rebuild the sparse matrix.

The overall structure of the aspect is given in *Figure 3.3*. The sparsity aspect identifies which variables are good candidates to make sparse by intercepting every set and get of every variable, and recording their size and sparsity. As a measure of sparsity we use ratio of nonzero to total number of elements in a matrix, i.e. the MATLAB expression `nnz(A)/numel(A)`. At the end of the program, a list of all variables along with the mean and standard deviation of their sizes and sparsities are printed out along with counts of accesses and shape as well as sparsity changes.

The existence of the get and set patterns are particularly convenient here, because we

```
aspect sparsity
  ...
  patterns
    arraySet : set(*);
    arrayWholeGet : get(*());
    arrayIndexedGet : get(*(..));
    exec : mainexecution();
  end

  actions
    ...
    aset : before arrayset : (newVal,obj,name)
      ...
    end

    awget : before arrayWholeGet : (obj,name)
      ...
    end

    aiget : before arrayIndexedGet : (args,name)
      ...
    end

    displayResults : after exec
    %display the results after the execution
  end
end
```

Figure 3.3 Outline of sparsity aspect

merely have to write actions in which we increase counters associated with every variable. Since the context information includes the name of a matched variable as a string, we can put all the variables in a MATLAB structure to map between names and values. A structure in MATLAB, unlike in static programming languages, allows the addition of fields during runtime. As new variables are encountered during runtime, they are added into the structure that tracks them, so we don't have to specify the variable names in advance. Thus, the aspect needs no modification to profile different programs.

Having special syntax allowing us to specify whether an array is accessed by indexing it or whether it is accessed without indexing allows us to differentiate between these accesses, and record them more easily.

Along with the aspect itself we coded our actual program, which utilizes a RungeKutta4 ODE solver [RLB05] to solve the heat equation in 1D given some initial conditions and time interval. The benchmark uses matrices to discretize the heat function in space. The needed derivative is computed using matrix multiplication with a differentiation matrix which is very sparse and never changes. Most of the computation of the program relies on this multiplication. If this matrix is made sparse, it decreases the overall computation time for this benchmark by 95% (tested in Matlab R2008a, on a linux PC with an AMD Athlon 64 X2 with 2GHz and 4GB of ram).

The output of the benchmark in *Figure 3.4* clearly shows that the variable D is of large size, never indexed, seldomly written or changing in shape or sparsity, but often used without indexes. We thus show a very simple benchmark using aspects and the special array patterns to profile a certain feature of a program, leading to a useful result. Without aspects and these patterns, one would need to inline profiling code manually.

```
>> program
tracking sparsities of all variables in the following program...
computation finished

’var’     ’size’              ’sparsity’   ’arraySet’   ’spty. inc.’  ’get’    ’ind. get’
’a’       ’1.0 0.0’          ’1.00 0.03’   [       2]   [        0]   [2002]   [       0]
’steps’   ’0.5 0.5’          ’1.00 0.00’   [       1]   [        0]   [   1]   [       0]
’tN’      ’0.5 0.5’          ’1.00 0.00’   [       1]   [        0]   [   1]   [       0]
’N’       ’1.0 0.1’          ’1.00 0.00’   [       3]   [        0]   [ 507]   [       0]
’h’       ’1.0 0.0’          ’1.00 0.00’   [       2]   [        0]   [3504]   [       0]
’X’       ’149.5 149.5’      ’1.00 0.00’   [       1]   [        0]   [   1]   [       0]
’U0’      ’199.3 140.9’      ’0.37 0.45’   [       2]   [        0]   [   1]   [       0]
’D’       ’89311.7 2823.6’   ’0.01 0.03’   [       3]   [        0]   [2000]   [       0]
’f’       ’0.0 0.0’          ’1.00 0.00’   [       1]   [        0]   [   0]   [       0]
’tspan’   ’1.3 0.9’          ’0.67 0.24’   [       1]   [        0]   [   0]   [       2]
’alpha’   ’149.5 149.5’      ’0.55 0.45’   [       1]   [        0]   [   1]   [       0]
’o’       ’0.0 0.0’          ’1.00 0.00’   [       1]   [        0]   [   0]   [       0]
’b’       ’0.7 0.5’          ’1.00 0.00’   [       1]   [        0]   [   2]   [       0]
’W’       ’149699.3 3863.3’  ’0.47 0.29’   [     504]   [      501]   [   1]   [    2500]
’t’       ’250.4 250.5’      ’1.00 0.00’   [    2001]   [        0]   [   0]   [    2000]
’j’       ’1.0 0.0’          ’1.00 0.00’   [     501]   [        0]   [6001]   [       0]
’u’       ’149.5 149.5’      ’0.98 0.09’   [    2000]   [        0]   [2000]   [       0]
’y’       ’0.0 0.0’          ’1.00 0.00’   [    2000]   [        0]   [   0]   [       0]
’k1’      ’298.8 7.7’        ’0.97 0.13’   [     500]   [        0]   [1000]   [       0]
’k2’      ’298.8 7.7’        ’0.97 0.13’   [     500]   [        0]   [1000]   [       0]
’k3’      ’298.8 7.7’        ’0.97 0.13’   [     500]   [        0]   [1000]   [       0]
’k4’      ’298.7 9.5’        ’0.97 0.13’   [     500]   [        0]   [ 500]   [       0]
```

Figure 3.4 Output of the sparsity benchmark

3.3 Measuring floating point operations

In numerical computing it is common to count computational complexity in terms of floating point operations, because they make up most of the operations. Knowing exactly how many floating point operations each part of a program performs can be more useful than knowing how much time the computations takes, because the number of flops may be more consistent, and is not subject to compiler optimizations.

The flop aspect, shown in *Figure 3.5*, thus attempts to identify where in the program floating point operations occur and counts them. For every occurrence of an operation on matrices (like times, mtimes, plus etc.), it uses an estimate on the number of floating point operations and records for every call site, the number of calls during the run of the program, and the total number of flops contained in all the calls.

This is done recursively, i.e., the output will list the total flops of a call of a function, but then it will also list the total flops for every call inside that function. This is done by keeping a stack that for every call records the number of operations performed so far. When encountering a new call, which is captured via a before action on all calls, zero is pushed onto the stack. When encountering a floating point operation, which is captured by using around advice for every tracked operation, the number of operations are added to the top of the stack. Finally, after every call, the number of operations encountered is added to the total operations of the call-site, and the operations are popped from the stack and is added to the next level.

Note that currently we have not defined patterns to match operations *, -, .*, etc., thus for this experiment such expressions have to be converted into their equivalent function form, i.e., using mtimes, minus, times, etc.

Note the order of the before and after actions. Because we want the “beforeTrack” and “afterTrack” to happen before and after the “any” actions, so that we can record information on the top level call that is being tracked, we have to list the actions in the order shown in *Figure 3.5*.

We used this aspect to weave into the computation of the singular value decomposition [Wat02]

```
aspect flops
  ...
  patterns
    tracking: call(SVD);

    pminus : call(minus (*,*));
    pmtimes : call(mtimes (*,*));
    ptimes : call(times (*,*));
    pplus : call(plus (*,*));
    psqrt : call(sqrt (*));
    prdivide: call(rdivide(*,*));
    pabs : call(abs (*));

    any : call(*);
  end % patterns

  actions
    beforeTrack : before tracking : (name)
      % before tracked call set up vars
    end

    bany : before any
      % before any call, take care of flops on stack(if recording)
      % push new 'stackframe' info
    end

    ... % put info on stack for every tracked operation

    aany : after any : (name,line,loc);
      % after any call, store info in variables and on 'stackframe'
    end

    afterTrack : after tracking
      % after tracked call print out results
    end
  end % actions
end % flops
```

Figure 3.5 Outline of flops aspect

of a random matrix. The utilized algorithm is spread over many files and operates on many 2x2 sub-matrices as well as the whole matrix, and it is not clear which operations dominate.

As the output in *Figure 3.6* shows, the aspect is able to uncover where most of the computation happens, and presents it in a similar way a profiler shows computation time (i.e., encapsulated information). In the listing, lines with 0 flops were removed for brevity.

While it would be possible in MATLAB to override the behavior of plus, minus etc (i.e., the atomic functions for which the aspect tracks the flops) to track the number of operations, it would be pretty much impossible to get that information in the way it is listed, i.e., with a report for every call site, and with encapsulated information, without emulating the before and after actions in some way.

3.4 Adding units to computations

The units aspect adds functionality by allowing matrices to have International System (SI) units associated with them, while not requiring any special treatment of these variables.

The outline of the aspect is given in *Figure 3.7*. It turns all variables that are encountered at calls into structures containing both a unit and the original value. All basic operations are overridden as well. In order to create a matrix with an associated unit, one merely has to multiply the matrix with the name of the unit.

The aspect intercepts all calls to functions that denote units (e.g. 's', 'Kg', 'inches', etc.), overrides them and returns a structure containing a value of one and the given unit. If the requested unit is not a basic SI unit (i.e., 'inches', 'kilotons') or if the value requested is a physical constant (i.e., 'AU', 'G', 'dozen') the value will be a factor relative to the corresponding SI unit. The point to note is that these functions that are getting called in a program don't exist anywhere on the MATLAB path. This is allowed in MATLAB, because if a name cannot be resolved an error is only thrown when the name is executed. But since we use an around action to intercept these calls, and replace them with the actual functionality they represent, they never get called by MATLAB. In effect, we use around actions to replace these "functions" with their real implementation.

```
>> runsvd
encountered call to SVD, recording flops...
finished tracking function call, here are the results:
   'call site'                 '# of calls' 'total flops'
   'fro_150_times'             [          1] [        100]
   'fro_150_sqrt'              [          1] [          1]
   'SVD_13_fro'                [          1] [        101]
   'SVD_14_abs'                [          7] [        630]
   'tinySVD_77_minus'          [        270] [        270]
   'tinySVD_77_plus'           [        270] [        270]
   'tinySVD_77_rdivide'        [        270] [        270]
   'tinySVD_78_sqrt'           [        270] [        270]
   'tinySVD_78_rdivide'        [        270] [        270]
   'tinySVD_79_times'          [        270] [        270]
   'tinySVD_81_mtimes'         [        270] [       3240]
   'tinySymmetricSVD_109_minus'  [        270] [        270]
   'tinySymmetricSVD_109_times'  [        270] [        270]
   'tinySymmetricSVD_109_rdivide'[        270] [        270]
   'tinySymmetricSVD_110_sign'   [        270] [          0]
   'tinySymmetricSVD_110_abs'    [        270] [        270]
   'tinySymmetricSVD_110_times'  [        270] [        270]
   'tinySymmetricSVD_110_plus'   [        540] [        540]
   'tinySymmetricSVD_110_sqrt'   [        270] [        270]
   'tinySymmetricSVD_110_rdivide'[        270] [        270]
   'tinySymmetricSVD_112_times'  [        270] [        270]
   'tinySymmetricSVD_112_plus'   [        270] [        270]
   'tinySymmetricSVD_112_sqrt'   [        270] [        270]
   'tinySymmetricSVD_112_rdivide'[        270] [        270]
   'tinySymmetricSVD_113_times'  [        270] [        270]
   'tinySymmetricSVD_116_mtimes' [        540] [       6480]
   'fixSVD_137_mtimes'         [        270] [       3240]
   'fixSVD_138_mtimes'         [        270] [       3240]
   'fixSVD_141_mtimes'         [         11] [        132]
   'fixSVD_142_mtimes'         [         22] [        264]
   'fixSVD_143_mtimes'         [         11] [        132]
   'tinySymmetricSVD_121_fixSVD' [        270] [       7008]
   'tinySVD_82_tinySymmetricSVD' [        270] [      17268]
   'tinySVD_83_mtimes'         [        270] [       3240]
   'jacobi_42_tinySVD'         [        270] [      25368]
   'SVD_17_jacobi'             [        270] [      25368]
   'SVD_18_mtimes'             [        540] [    1026000]
   'SVD_19_mtimes'             [        270] [     513000]
   'SVD_20_mtimes'             [        270] [     513000]
   'SVD_34_times'              [          1] [        100]
   'Script_6_SVD'              [          1] [    2078199]
```

Figure 3.6 Output of the flops benchmark

```
aspect unit
 ...
 patterns
  disp : call(disp);
  plus : call(plus(*,*));
  minus : call(minus(*,*));
  mtimes : call(mtimes(*,*));
  mrdivide : call(mrdivide(*,*));
  power : call(power(*,*));
  round : call(round(*));
  colon : call(colon(*,..));

  allCalls : call(*());

  loopheader : loophead(*);
 end
 ...
 actions
  ... % overwrite all operations and annotate

  loop : around loopheader : (newVal)
   range = this.annotate(newVal);
   acell = {};
   for i = (range.val)
    acell{length(acell)+1} = i;
   end
   varargout{1} =
   struct(this.annotated,true,'val',acell,'unit',range.unit);
  end
 end % actions
end % unit
```

Figure 3.7 Outline of units aspect

All operations (again only the functions, not the operators) are overridden to both perform the denoted operation on the .val field and the .unit field. Units are stored as vectors, denoting the power of every SI unit. There are 7 SI units, and they are ordered as metre, kg, second, Ampere, Kelvin, candela and mol. Thus, [1 0 -2 0 0 0 0] would denote m/s^2. The function 'dis' is overridden as well to show the matrix with the associated unit.

Because the data structures MATLAB now computes with are changed, all the semantics in the program change. In particular, for loops using the syntax

```
for i = x
```

```
t = mrdivide(AU,c);
%t = AU/c;
disp(t);
% bmi given in imperial units
bmi =
  mtimes(180,mrdivide(lb,
      ower(plus(mtimes(5,feet),mtimes(8,inches)),2)));
% bmi = 180*lb/(5*feet + 8*inches)^2
disp(bmi);
```

Figure 3.8 Example of units aspect

do not work anymore, because `x` will no longer be an array, but instead it will be a structure containing an array in the field "val". Thus we use the loophead pattern and override the loop initialization, to turn the array into a struct-array. The struct-array is a MATLAB array whose every element, when indexed, is a structure. This data type works with for loops again, allowing us to emulate the correct semantics.

In *Figure 3.7*, the action takes the range expression, and iterates over the values. These are stored in a cell array, which is then passed to the struct function which creates a structure array. This is a feature of MATLAB- when 'struct' receives a cell array, it will build a struct-array. When looping over this new structure, every element will be a structure containing the elements of value of the previous array.

For example, one could run the code given in *Figure 3.8*, for which the result after weaving and running would be:

```
s: 499.0052
m^-2*Kg: 27.3686
```

This example demonstrates that AspectMatlab allows us to override the functionality of matrices, adding support for numerous units, adding a language extension supporting many of the basic operations while keeping the semantics, all with an aspect that is less than 300 lines long.

3.5 Interpreting loop iteration space

This aspect extracts the **for** loop iteration space and then interprets this space within the loop body. It can be useful in many applications, for example, for iterative solvers that get called a lot to see how many iterations are performed, or **for** loop dependency analyses where we need to know the lower bound, upper bound and the increment factor for the **for** loops. This benchmark can also be used to track how long loops run throughout the program.

An outline of the aspect is shown in *Figure 3.9*. This aspect consists of several patterns to first extract the loop iteration space using the `loophead` and the `loopbody` patterns. Within the loop body, a programmer can inquire about the different attributes of the iteration space by using the call expression. So the second set of patterns in this aspect are the **call** patterns, along with **within** patterns to restrict the scope to the loop body.

Action `aLoopHead` is called right after the evaluation of the loop iteration assignment statement and it keeps the iteration array. This information is pushed into a stack maintained in aspect properties. Action `aLoopBody` acts at the beginning of the each iteration and updates the current iteration counter of the current top entry in the stack. Action `aLoop` is called after the loop, and it is used to pop the entry of the loop from the stack.

The rest of the actions actually replace the calls made to fetch loop iteration space from within the loop. For example, when a user wants to fetch the current iteration number in a loop, an imaginary call can be made to `iteration` function. This aspect will capture such calls made from within a loop and return the current iteration number for that particular loop. Other information that can be asked for is lower bound (`lBound`), upper bound (`uBound`), and increment factor (`increment`). Its important to note here that its not possible to have an increment factor for all loop iteration spaces. For example, if the iteration array consists of random numbers in no specific sequence, then there is no concept of increment and a MATLAB-specific `NaN` value is returned.

Consider the example given in *Figure 3.10* which consists of a loop iterating on `i` and a nested loop iterating on `j`. It would generate the output shown in *Figure 3.11*, for the 19th

```
aspect loops
  ...
  patterns
    plhead : loophead(*);
    plbody : loopbody(*);
    ploop : loop(*);

    lbound : call(lBound) & within(loops,*);
    ubound : call(uBound) & within(loops,*);
    increment : call(increment) & within(loops,*);
    iteration : call(iteration) & within(loops,*);
  end % patterns

  actions
    aLoopHead : after plhead : (newVal)
      % extracts the loop iteration space
      % push the entry on the stack
    end

    aLoopBody : before plbody : (counter)
      % extracts the loop iteration
      % update the current top on the stack
    end

    aLoop : after ploop
      % pop the top entry from the stack
    end

    aLBound : around lbound
      % returns the lower bound for the loop
    end

    aUBound : around ubound
      % returns the upper bound of the loop
    end

    aIncrement : around increment
      % returns the increment factor of the loop
    end

    aIteration : around iteration
      % returns the current iteration number of the loop
    end
  end % actions
end % flops
```

Figure 3.9 Outline of loops aspect

```
 for i = 1:2:99
  disp({'Lower Bound i: ', lBound});
  disp({'Upper Bound i: ', uBound});

  for j = 10:-1:1
    disp({'Lower Bound j: ', lBound});
    disp({'Upper Bound j: ', uBound});
    disp({'Increment j: ', increment});
    disp({'Current Iteration j: ', iteration});
  end

  disp({'Increment i: ', increment});
  disp({'Current Iteration i: ', iteration});
end
```

Figure 3.10 Example of loops aspect

```
> ...
  Lower Bound i: 1
  Upper Bound i: 99
  ...
  Lower Bound j: 10
  Upper Bound j: 1
  Increment j: -1
  Current Iteration j: 5
  ...
  Increment i: 2
  Current Iteration i: 19
  ...
```

Figure 3.11 Output of loops aspect

iteration of outer loop and the 5th iteration of the nested loop, skipping the ouput of the other iterations.

3.6 Other possibilities

While we only presented a few use cases showcasing the potential of both aspects in the scientific computing domain as well as our special patterns in particular, there are many more possibilities.

For example, one could use the loop patterns to track how many iterations an iterative solver like iterative GEPP or solver based on the Newton Method. This could be particularly useful if it is used inside some larger computation like a backward Euler integration, because it would allow one to track how many iterations are done when and where.

Tracking loop counts could also be interesting for loop dependency analyses. One could use aspects there to collect run time information and feed that back to the compiler and write specializing code optimizing the encountered runtime properties. This could be done for many possible optimizations.

Another interesting aspect that overrides all possible values could be one that overrides all computation with equivalent operations utilizing interval-arithmetic. Variables could be initialized to small intervals corresponding to the unit precision, which get larger as more computation is performed. The advantage would be that interval arithmetic gives hard bounds on the computed values so that there are no surprises due to rounding arithmetic. One could also use more simplified runtime forward error propagation schemes.

AspectMatlab provides some interesting ways to combine numerical computing with aspect-oriented programming, resulting in some of the use cases shown. Many other possibilities are conceivable, which motivate the development of the language in general and the array and loop patterns in particular.

Chapter 4
Compiler

The AspectMatlab compiler (*amc*) has been designed to be easily extensible so that it is simple for us and other researchers to add further features. To enable this we have built the compiler using extensible toolkits and have aimed for a very clean and modular implementation.

This chapter examines the design of our AspectMatlab compiler in detail. We begin with a discussion of the overall architecture of the compiler and an overview of its different phases. This is followed by a detailed discussion of each of the phases of the compiler. We start with a discussion on how the front-end tools enabled us to create the extension. We describe a set of transformations on the source code that are required in order to perform accurate matching and weaving. Then we discuss the name resolution analysis which is used to eliminate most of the dynamic checks introduced by the weaving phase. Finally, we conclude with a detailed example of a woven aspect and a discussion on the performance overheads introduced by the aspect woven code.

4.1 Compiler Structure

The overall structure of *amc*, the AspectMatlab compiler, is given in *Figure 4.1*. The compiler takes as input, a collection of MATLAB (.m) source files, plus a collection of

AspectMatlab files, and produces a collection of woven MATLAB source files. These output files can be executed using any MATLAB system.

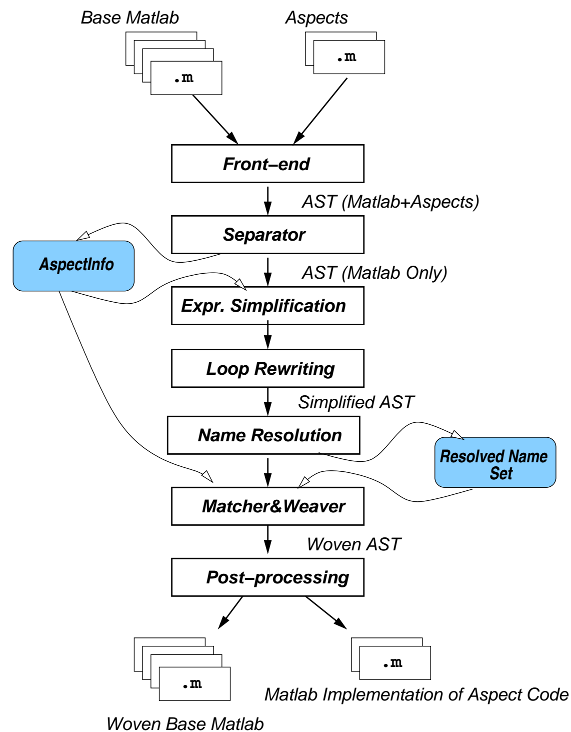


Figure 4.1 Overall structure of the *amc* AspectMatlab compiler

The front-end of AspectMatlab was implemented as an extension to the Natlab front-end (Natlab is a ”neat” version of MATLAB, developed by the Sable Research Group). The scanner is built using the MetaLexer tool [Cas09] and was specified as a simple and modular extension to the Natlab Metalexer specification. The parser and semantic checks were modular extensions to Natlab's parser, which is built using the extensible JastAdd framework[EH07]. The Natlab grammar was extended to incorporate AspectMatlab grammar rules using the JastAddParser. JastAdd provides powerful facilities for AST traversal, associating attributes with nodes and modifying the AST via node rewriting.

4.2 Separator & AspectInfo

As indicated in *Figure 4.1*, after front-end processing, the AST generated includes both MATLAB and aspect-specific AST nodes. Following the *abc* model, the Separator component harvests all the aspect-specific key information out of the AST, and transforms the AST so that it becomes a pure MATLAB AST. This process allows us to process the resulting AST using our Matlab compiler analysis framework, and is also the first key step in converting the aspect source files to MATLAB source files.

The separation phase records the aspect information into a collection of data structures, called *AspectInfo*. *AspectInfo* consists of several data structures, which are used to contain the aspect lists, pattern lists, action lists and the information about their association.

AspectInfo contains the following structures:

- Pattern lists encode simple mappings of the pattern designator given names to the actual pattern designator expressions. These lists are used to perform expression simplification (*Section 4.3.1*) before matching.

- An action defined in the aspect files is translated into a normal MATLAB function and related information is stored in an object called *ActionInfo*. It keeps the action name, associated pattern designator name, type of the action, a reference to the translated function corresponding to the action and the host aspect name. Action lists are lists of all such objects. In order to match and weave, both action lists and their mapping to corresponding patterns are used (*Section 4.5*).

- *AspectInfo* also keeps a simple list of all the aspect files presented to the compiler. This list of aspects is mainly used in the post-processing phase, which we will discuss in *Section 4.6*.

4.3 Transformations

In order to perform matching and later weaving, some join points require transformation. There are two notable code transformations: name/parameterized expressions simplification and loops rewriting.

4.3.1 Expression Simplification

An expression in MATLAB can be very complex with a lot of computation being performed within a single expression. This computation can be in the form of function/script calls or complex operations on arrays. So some kind of refactoring of complex expressions is required to expose all the matching and weaving points in the code. To avoid inserting meaningless and redundant code, we consult the AspectInfo data structures at this stage. All the name or parameterized expressions, which can potentially match the specified patterns, are taken out of the parent expression. An expression in the source code is a potential match, if there exists a pattern in the pattern list fetched in AspectInfo. An important point to notice here is that a pattern can be used by an action, either solely or in combination with other patterns, or none of the actions end up using a particular pattern. So, at this stage we go through all the patterns and simplify the complex expressions in order to facilitate matching and weaving later. This results in simple weavable statements with precise locations for before, after or around actions.

For example, given the following line in the base program:

```
1  z = sum(x) / length(y);
```

and assuming that there is only one pattern for call to `sum`, the above line gets translated into this:

```
1  AM_CVar_1 = sum(x);
2  z = AM_CVar_1 / length(y);
```

Now assuming that patterns exist for both calls and variable accesses, the above line gets translated into the following:

```
AM_CVar_1 = x;
AM_CVar_2 = sum(AM_CVar_1);
AM_CVar_3 = y;
AM_CVar_4 = length(AM_CVar_3);
z = (AM_CVar_2 / AM_CVar_4);
```

It should be noted that arguments of the function calls are extracted out for the sake of their own weaving. So the arguments always get evaluated before the function call. In the case of a **before** action or an **after** action, it doesn't change the semantics of a function call. In turn, in the **around** case, if the function never gets called through **proceed**, its arguments would still be evaluated before passed on to the **around** action.

4.3.2 Loop Rewriting

The second kind of transformation occurs on the loops. In MATLAB **for** loops have a loop iteration space defined before the loop executes - **for** loops contain an assignment statement, which allocates the iteration space to the loop iterator. In order to perform weaving on that assignment statement itself, it needs to be taken out of the loop body and be replaced by appropriate code.

For example, consider the following loop:

```
for i=1:step:size(dx,1)
  % loop body
end
```

This loop would be transformed to the following. Note that **for** loops are transformed regardless of the existence of any patterns or actions targeting them, for reasons we shall describe in the following section.

```
AM_CVar_5 = 1:step:size(dx,1);
for AM_CVar_6 = 1:length(AM_CVar_5(:,:))
  i = AM_CVar_5(:,AM_CVar_6);
  % loop body
end
```

A different challenge is presented by **while** loops. The conditional expression can contain several instructions inside it. Refactoring the expression will be our solution again. But

since the condition is supposed to be evaluated at the start of each iteration, we have to take care of all the back edges of the loop finishing at the loop header, which means just before the syntactic end of loop body and also at all the `continue` statements.

For example, consider the following loop.

```
while x < y
  % ...
  continue;
  % ...
  % loop body
  % ...
end
```

In the transformed version below, the **`while`** loop's conditional expressions are factored out and placed before all the edges in the loop header.

```
AM_CVar_7 = x < y;
while AM_CVar_7
  % ...
  AM_CVar_7 = x < y;
  continue;
  % ...
  % loop body
  % ...
  AM_CVar_7 = x < y;
end
```

4.4 Name Resolution Analysis

In MATLAB, a function call or an array access has the same syntax using either just the name of the entity or passing a number of parameters with it. So `foo(1, 2)` can either be an access to an array named `foo`, if the array exists in the current scope, or it could be a function call with two parameters. This name resolution can be achieved with the help of runtime checks, but doing so we compromise on the efficiency of generated woven code.

Figure 4.2 shows an aspect which contains two different kinds of patterns targeting the same entity *foo* in the source code, i.e., **`get`** and **`call`**. There are two **`before`** actions matching on both patterns respectively.

```
1  aspect nameRes
2
3  %properties
4
5  %methods
6
7  patterns
8    pGetFoo : get(foo);
9    pCallFoo : call(foo);
10 end
11
12 actions
13   aGetFoo : before pGetFoo
14     % action body
15   end
16
17   aCallFoo : before pCallFoo
18     % action body
19   end
20 end %actions
21
22 end %aspect
```

Figure 4.2 Aspect with multiple patterns on the same entity

When this aspect is applied to a source code containing the following line:

```
x = foo(3);
```

The entity foo needs to be resolved as a function call or an array access in order to be accurately matched with the patterns given in the aspect. So we need to insert dynamic checks for each action at the point of weaving, as shown in *Figure 4.3*.

```
1   if (exist('foo', 'var') == 1)
2     AM_GLOBAL.nameRes.nameRes_aGetFoo();
3   end
4   if (exist('foo', 'var') ~= 1)
5     AM_GLOBAL.nameRes.nameRes_aCallFoo();
6   end
7   x = foo(3);
```

Figure 4.3 Weaving without Name Resolution Analysis

The call to the aspect action depends on the outcome of the dynamic check, i.e., if foo is a variable then call to aGetFoo action is made, aCallFoo action is called if foo happens to be a function.

But once we have all the names resolved in the source code, the weaving process becomes very simple. As shown in *Figure 4.4*, if foo is resolved as an array at the joint point then only the **get** pattern is matched and the **call** pattern is not considered. Vice versa in the case of foo being a function.

```
%if foo is known to be an array at this point
AM_GLOBAL.nameRes.nameRes_aGetFoo();
x = foo(3);

-----------------OR------------------

%if foo is known to be a function at this point
AM_GLOBAL.nameRes.nameRes_aCallFoo();
x = foo(3);
```

Figure 4.4 Weaving with Name Resolution Analysis

So we essentially need to have a flow analysis, to determine the exact type of a join point at the time of matching[1]. The goal of this analysis is to determine if a given name at a given program point corresponds to a function, variable or assigned variable. To accomplish this goal, the analysis is implemented as a data flow analysis using the McLab Analysis Framework. This flow analysis builds up a set of information for each statement in the program, that we call resolved names set. This set contains names labeled with the information about those names.

The MATLAB semantics for determining if a name is a variable or a function within function bodies are fairly static. Because of this, the Name Resolution Analysis is capable of accurately determining all names that are variables. This allows the compiler to eliminate all runtime checks for this property. By eliminating those checks the compiler also eliminates all uses of eval. The analysis can also be fairly successful in eliminating runtime checks to determine if a variable is defined in a function. Once again this is due to the more

[1]This analysis was developed as part of Jesse Doherty's Masters thesis

static semantics of functions. The analysis can determine that roughly half of the variable uses in our example programs given in *Chapter 3* are guaranteed to be well defined. Script semantics are more dependent on runtime behaviour. Because of this the Name Resolution Analysis is less successful at determining how names are resolved in scripts bodies.

4.5 Matching and Weaving

The previous name resolution phase populates the resolved names set, which is then used as one input to the matcher and weaver, along with pure MATLAB AST and *AspectInfo* structure.

An outline of the matching and weaving process is shown in *Figure 4.5*. In a single pass through the AST, all the join points are matched against the patterns specified for all the actions. In case a shadow of a join point matches a pattern designator, the corresponding action is woven at the appropriate place with respect to the location and type of the shadow.

```
for each base file
  for each join point j
    s = j.getShadow();
    for each action a
      p = a.getPattern();
      if(p.match(s))
        s.weave(a);
```

Figure 4.5 Matching and Weaving process outline

4.5.1 Weaving at the function level

AspectMatlab provides an `execution` pattern to match at the level of the functions and the scripts. Since both the functions and the scripts are named entities, matching by name is straight forward. The matching actions are woven as a call to the action member function of the class generated from the aspect definition. All the `before` actions are woven in order just above the first statement in the body, and all the `after` actions go right below

the last statement, or just before all the `return` statements. In case of the `around` of `execution` (and other kinds of patterns as well), the semantics of MATLAB force us to develop a different strategy, which is described in section 4.5.4.

4.5.2 Weaving at the loop level

AspectMatlab provides a set of loop patterns for both **`for`** and **`while`** loops, namely `loop`, `loopbody` and `loophead`. Unlike other program constructs, loops are not named entities. So we match the loops based on the variables involved inside the loop header. There is a single loop iteration variable in **`for`** loops, whereas the conditional expression of the **`while`** loop can contain any number of named entities. The question might arise here that names for loops iterator variables are often very general (for example, i or j), so we might end up over-matching loops unintentionally. The **`within`** pattern comes in very handy in such situations to restrict the scope of matching to specific constructs.

Weaving **`for`** loops is also different than **`while`** loops with regard to the context information they provide. We can fetch the loop iteration space (out of which the action function can infer start, end and stride values of the loop iterator), loop iterator variable and loop counter. In order to weave an **`after`** action on a `loopbody` pattern, we have to analyze the loop body, because it's not just the syntactical end of the body. We also have to take **`break`**, **`continue`** and **`return`** statements into account, as they mark the end of body too.

For the `loop` join point, all the `before` actions are woven in order just above the loop, and all the `after` actions go right below the loop. In case of the `loopbody` join point, all the `before` actions are woven in order just above the other statements in the body, and all the `after` actions go right before the end of the loop body, or before any of the **`break`**, **`continue`** and **`return`** statements. Because loop headers were translated into separate statements in an earlier phase, they get matched and woven just like other statements as described in *Section 4.5.3*.

4.5.3 Weaving at the statement level

Since we have simplified the complex statements already in an earlier pass, it only comes down to assignment statements or even simple expression statements. The left hand side of an assignment statement is matched for `set` patterns, and right hand side expression for `get` or `call` patterns. This is where we can use the name resolution set, which helps us determine if the expression is a `get` or `call` join point. Without the name resolution optimizations we must weave in a dynamic check. All the `before` actions are woven in order just above the statement, and all the `after` actions go right below the statement.

4.5.4 Weaving `around` actions

In the AspectJ around advice case, the concerned piece of code is extracted out of the context and replaced with a call to the around advice. The extracted code is placed inside a new method of the same class, which is then called from aspect's advice function. Because the code stays in the same class, there are no scoping issues. However, in the case of MATLAB's non object-oriented version, this weaving strategy is clearly not possible. When we move a piece of code out of its scope, we have to provide all the necessary context information required.

The solution we came up with is partially inspired by Kuzins' work on efficient implementation of around advice for the AspectBench Compiler [Kuz04]. Taking advantage of the MATLAB's nested functions, we create a nested function, namely **`proceed`**, inside the around action function. This function contains a **`switch`** statement to host the extracted code from all the around join points of this particular action. The join points are assigned a simple number id, one id for each around action. Along with this id, a join point has to pass the context information to execute the extracted code being moved inside a different scope.

The translated standard MATLAB function for the **`around`** action from the example given in *Figure 2.9* is shown in *Figure 4.6*. The first thing to notice here is that the **`proceed`** function is created inside the **`around`** function as a sub-function. As explained earlier, this function

```
1  function [varargout] = myAspect_actcall(this, AM_caseNum, AM_obj,
      AM_args, name, args)
2    this.incCount();
3    disp(['calling ', name, 'with parameters(', args, ')']);
4    proceed(AM_caseNum, AM_obj, AM_args);
5
6    function [] = proceed(AM_caseNum, AM_obj, AM_args)
7      switch AM_caseNum
8        case 0
9          varargout{1} = AM_obj(AM_args{1}, AM_args{2});
10       case 1
11         varargout{1} = AM_obj(AM_args{1}, AM_args{2});
12       case 2
13         AM_obj(AM_args{1}, AM_args{2});
14     end %switch
15   end %proceed
16 end
```

Figure 4.6 Example of an **around** function

builds switch cases for all the join points matched for this particular action. If applied to the code given in *Figure 2.10*, this action matches at lines 5, 6 and 9. Accordingly there are three switch cases added to the **proceed** function. `AM_obj` contains the actual object from the shadow, which can either be a variable or a function name. `AM_args` represents the actual arguments used at the shadow.

AspectMatlab supports the concept of multiple **around** actions. As shown is *Section 2.3.3*, multiple **around** actions are woven around the join point in the exact order in which actions are defined in source code. So in case of multiple **around** actions, the actual join point is executed in the very last action, and all other actions just go around each other in order. *Figure 4.7* shows an aspect which contains two **around** actions on the same pattern, which matches all calls made to function `foo`.

Consider this aspect is applied to a source code containing the following line:

```
x = foo(3);
```

Figure 4.8 shows the translated MATLAB functions for the **around** actions. The **proceed** in `multiAround_actAround1` only contains a call to `multiAround_actAround2`,

```
1  aspect multiAround
2
3  %properties
4
5  %methods
6
7  patterns
8    pCallFoo : call(foo);
9  end
10
11 actions
12   actAround1 : around pCallFoo
13     disp('before around action 1');
14     proceed();
15     disp('after around action 1');
16   end
17
18   actAround2 : around pCallFoo
19     disp('before around action 2');
20     proceed();
21     disp('after around action 2');
22   end
23 end %actions
24
25 end %aspect
```

Figure 4.7 Aspect for multiple `around` actions

which actually executes the actual join point through its `proceed`. All context info is passed from the join point shadow in case of `around` actions.

If the call to function `foo` just prints out the number passed to it, then the output of the multiple `around` actions would be:

```
> before around action 1
  before around action 2
  foo 3
  after around action 2
  after around action 1
```

```
function [varargout] = multiAround_actAround1(this, AM_caseNum, AM_obj,
    AM_args, [context info])
  disp('before around action 1');
  proceed(AM_caseNum, AM_obj, AM_args);
  disp('after around action 1');

  function [] = proceed(AM_caseNum, AM_obj, AM_args)
    switch AM_caseNum
      case 0
        varargout{1} = this.multiAround_actAround2(AM_caseNum, AM_obj,
            AM_args, name, args);
      % Other cases
    end %switch
  end %proceed
end

function [varargout] = multiAround_actAround2(this, AM_caseNum, AM_obj,
    AM_args, [context info])
  disp('before around action 2');
    proceed(AM_caseNum, AM_obj, AM_args);
    disp('after around action 2');

  function [] = proceed(AM_caseNum, AM_obj, AM_args)
    switch AM_caseNum
      case 0
        varargout{1} = AM_obj(AM_args{1}, AM_args{2});
      % Other cases
    end %switch
  end %proceed
end
```

Figure 4.8 Translated multiple `around` functions

4.6 Post-processing

At the end of the weaving, a post-processing phase takes place. As explained earlier, the aspect files are translated into classes. So the aspect actions are woven as a call to a corresponding class methods. These class objects should be instantiated at the program entry point. As compared to AspectJ where Java provides a designated entry point through `main`, MATLAB does not has the same feature. Using the MATLAB interpreter, a user can choose any entry point for the woven program files, including function lists, scripts and classes.

So unless a user designates a function or a script to be the starting point of execution at the time of weaving[2], we need to embed the startup checking code at the start of all functions and scripts. As it can be seen in *Figure 4.10*: lines 2-8, these checks determine the very first function or script to be executed and then instantiate all the aspect objects. These objects need to be live and accessible throughout the execution environment over several program files, so we save them in a global structure, called `AM_GLOBAL`. In all the woven functions and scripts, declaration of this global variable is woven as their first statement. At the program exit point, the contents of the global variable are cleared to start afresh next time, *Figure 4.10*: lines 33-36. Finally, *amc* generates standard MATLAB code which can be executed by any MATLAB system.

4.7 Woven Example

After this detailed description of all phases, we come back to the example given in *Section 2.4*, where we used a simple aspect to count the function calls. The aspect itself is translated into a standard MATLAB class file, as shown in *Figure 4.9*. The class extends the MATLAB built-in *handle* class, which enforces this subclass to be a reference class. Reference classes in MATLAB use a handle to reference to multiple objects of the class, as compared to the value classes where a new object is always created in case of the copy operation.

It can be noticed that the generated class preserves the properties and methods blocks from the aspect. The patterns block gets eliminated, and the all the actions are translated into standard MATLAB class methods.

The woven code for the example given in *Section 2.4* is shown in *Figure 4.10*. Expression simplification is very noticeable, as we transform complex statements into easy-to-weave statements. With the help of our name resolution analysis, we are able to distinguish between function calls and array accesses. All the calls matching the `call` pattern are woven accordingly. Notice the extra bit of code added by the post-processing phase, at the start and the end of the function.

[2] `java -jar amc.jar -main myFunc1.m myFunc2.m myAspect.m`

```
classdef myAspect < handle
  properties
    count = 0;
  end
  methods
    function [out] = getCount(this)
      out = this.count;
    end
    function [] = incCount(this)
      this.count = (this.count + 1);
    end
  end
  methods
    function [varargout] = myAspect_actcall(this, AM_caseNum, AM_obj,
        AM_args, name, args)
      this.incCount();
      disp(['calling ', name, 'with parameters(', args, ')']);
      proceed(AM_caseNum, AM_obj, AM_args);
      function [] = proceed(AM_caseNum, AM_obj, AM_args)
        switch AM_caseNum
          case 0
            varargout{1} = AM_obj(AM_args{1}, AM_args{2});
          case 1
            varargout{1} = AM_obj(AM_args{1}, AM_args{2});
          case 2
            AM_obj(AM_args{1}, AM_args{2});
        end
      end
    end
    function [] = myAspect_actexecution(this)
      count = this.getCount();
      disp(['total calls: ', num2str(count)]);
    end
  end
end
```

Figure 4.9 MATLAB class generated from the aspect

```
1  function [m, s, d] = histo(n)
2    global AM_GLOBAL;
3    if (isempty(AM_GLOBAL))
4      AM_EntryPoint_0 = 1;
5      AM_GLOBAL.myAspect = myAspect;
6    else
7      AM_EntryPoint_0 = 0;
8    end
9    AM_CVar_0 = n;
10   AM_CVar_1 = AM_GLOBAL.myAspect.myAspect_actcall( 0, @randn,
       {AM_CVar_0, 1}, 'randn', {AM_CVar_0, 1});
11   % Generate vectors of random inputs
12   % x1 = Normal distribution N(mean=100,sd=5)
13   % x2 = Uniform distribution U(a=5,b=15)
14   x1 = ((AM_CVar_1 * 5) + 100);
15   AM_CVar_2 = n;
16   AM_CVar_3 = AM_GLOBAL.myAspect.myAspect_actcall(1, @rand, {AM_CVar_2,
       1}, 'rand', {AM_CVar_2, 1});
17   x2 = (5 + (AM_CVar_3 * (15 - 5)));
18   AM_CVar_4 = x2;
19   AM_CVar_5 = x1;
20   y = ((AM_CVar_4 .^ 2) ./ AM_CVar_5);
21   AM_CVar_6 = y;
22   AM_GLOBAL.myAspect.myAspect_actcall(2, @hist, {AM_CVar_6, 50},
       'hist', {AM_CVar_6, 50});
23   AM_CVar_7 = y;
24   AM_CVar_8 = mean(AM_CVar_7);
25   % Calculate summary statistic
26   m = AM_CVar_8;
27   AM_CVar_9 = y;
28   AM_CVar_10 = std(AM_CVar_9);
29   s = AM_CVar_10;
30   AM_CVar_11 = y;
31   AM_CVar_12 = median(AM_CVar_11);
32   d = AM_CVar_12;
33   if AM_EntryPoint_0
34     AM_GLOBAL.myAspect.myAspect_actexecution();
35     AM_GLOBAL = [];
36   end
37 end
```

Figure 4.10 Woven MATLAB function

4.8 Performance Overhead

In this section, we briefly discuss the performance overhead introduced by the woven code. We performed a comparative execution of the benchmarks discussed in *Chapter 3*. We tested in Matlab R2008a, on a linux PC with an AMD Athlon 64 X2 with 2GHz and 4GB of RAM.

The results are shown in Table 4.1. All the times are given in seconds. It is worth mentioning that the first two benchmarks operate mostly on the matrices, whereas the other two benchmarks operate mostly on scalars. With the introduction of the aspects, we timed the resultant woven code in two ways: without the action code and then with the action code. Timing the woven code without the action body gives us the idea of the slowdown factor introduced purely by the aspect action calls.

We seem to get a slowdown with factor 1.23 to 2.72 in four benchmarks. The number is very high with counting the floating point operations bechamrk, 47.69, because this particular aspect was used to weave into an algorithm for the computation of the singular value decomposition of a random matrix. The woven algorithm, spread over multiple files, triggers an action call for each operation, hence resulting in higher slowdown.

So the aspect overhead is largely due to a couple of factors; (1) these benchmarks cross-cut on all the arrays being set or being read, or all the function calls being made, which means frequent calls to the aspect actions, and (2) the large matrices are being inquired by the action code as part of the context information, and the fact that MATLAB creates the copies of all the arrays being assigned or modified inside a function, when passed as an argument, resulting in the performance slowdown.

We believe that the performance overhead due to the aspects can be remarkably reduced with the help of code in-lining, as it is proven by AspectJ. The implementation of a copy elimination analysis can also play a vital role, if we can pass the large arrays just by reference instead of making copies.

Benchmark	Run time	Woven run time (without action code)	Woven run time (with action code)	Slowdown factor
Grow	3.463019s	9.414565s	18.035407s	2.72
Sparsity	3.761089s	5.733333s	40.175475s	1.52
Flops	0.057600s	2.746667s	19.364130s	47.69
Units	0.040411s	0.071931s	23.769422s	1.78
Loops	0.025776s	0.031704s	0.626242s	1.23

Table 4.1 Performance overhead

Chapter 5
Related Work

AspectMatlab is targeted at dynamic scientific programs, and thus deals with a different set of challenges as compared to other aspect-oriented language extensions. In this chapter, we review a number of such works, and contrast them with the approach taken in AspectMatlab.

We begin with the most popular aspect-oriented system called AspectJ, an extension to Java. Basic concepts and constructs of AspectMatlab are mostly inspired by AspectJ, though customized for MATLAB semantics. We discuss AspectJ and a set of extensions to AspectJ that were directly related to our work in *Section 5.1*.

Our research is also inspired by another aspect-oriented language, called AspectCobol. We discuss the similarities of AspectCobol to our work in *Section 5.2*. We conclude with a brief reference to another effort made to introduce aspects to MATLAB.

5.1 AspectJ

AspectJ [KHH+01] was one of the main languages that popularized aspect-oriented programming. AspectJ provides array pointcuts functionality, such that a type name pattern or subtype pattern can be followed by one or more sets of square brackets to make ar-

ray type patterns. So Object[] is an array type pattern, as is com.xerox..*[][] and Object+[]. However, the pointcuts of AspectJ do not support array objects in full. When an element of an array object is set or referenced, the corresponding index values and the assigned value are not exposed to the advice. The availability of such information can be very helpful in multiple ways, such as the ability to bounds-check the array, optimization of array usage and profiling related to arrays. The original AspectJ does not support any loop pointcuts.

Researchers have experimented with array and loop pointcut extensions to AspectJ using abc, an extensible AspectJ compiler [ACH+05].

5.1.1 Extension: Array specific pointcuts

Harbulot extended the set pointcut to capture arrayset.[1] In his proposal, the pointcut designator args() exposes both the array index value and the object being assigned to an array element, and the pointcut designator target() exposes the array object being assigned. However, this extension bases its implementation on treating an array element set as a call to a set(int index, Object newValue) method, and thus works only for one-dimensional arrays.

5.1.2 Extension: Multi-dimensional array specific pointcuts

As compared to Harbulot's extension, ArrayPT [CC07] works for multi-dimensional arrays. The core of the implementation is a finite-state machine based pointcut matcher that can handle arrays of multiple dimensions in a uniform way. They took the standard field set pointcut as the basis and developed this extension on the top of it. All array field set join points are treated as having a variable number of arguments: the sequence of index values and the value the field is being set to. At a join point, these values can be obtained using an args() pointcut designator and then passed to the advice for further processing. It enables the programmer to perform selective matching on any number of specified indices.

[1] Post to the abc-users mailing list, November 2004.

```
aspect Monitor {
 before(int ix1, int ix2, int newVal):
    arrayset(* Watch.*) &&
    args(ix1, ix2, newVal) { //advice
   if (newVal > B.bounds[ix1, ix2]) {
    ArraySetSignature sig =
        (ArraySetSignature)thisJointPoint.getSignature();
    String field = sig.getFieldType() + sig.getName();
    throws new RuntimeException("Bad change"+ field)
   }
  }
}
```

Figure 5.1 Example of AspectJ multi-dimensional array pointcuts (from [CC07])

For example, the aspect given in *Figure 5.1* uses the arrayset() and args() pointcut designators to monitor the assignments to any array fields of class Watch. Notice the use of the pointcut designator `args(ix1, ix2, newVal)` to get the array index values and the assigned value of the array field assignments. Here `args(ix1, ix2, newVal)` also serves as the selective matching, because it makes the aspect match only on the assignments on the arrays with two indices.

AspectMatlab enhances this idea of selective matching and incorporates it within the definition of a pattern designator. It eliminates the need of a separate pattern for capturing arrays and then using another pattern to specialize the matching. AspectMatlab also can more easily detect array set and get join points as it matches at the source code level, whereas the AspectJ approaches all must match and weave at the Java bytecode level.

5.1.3 Extension: Loop specific pointcuts

Another extension to the abc compiler, LoopsAJ [HG06], provides AspectJ with a loop pointcut. Loop selection is a major issue here, because unlike other pointcuts for variables and functions, loops don't have a named identification associated with them. In AspectMatlab, loop patterns are equipped with a facility to match the loops based on the variables being used in loop headers. Certain context exposure is provided to make the advice more effective.

```
void around(int min , int max , int step ):
    loop() && args (min, max, step, ..) {
  int numThreads = 4;
  Thread[] threads = new Thread[numThreads];
  for (int i = 0; i < numThreads; i++) {
    final int t_min = min+i;
    final int t_max = max;
    final int t_step = numThreads * step;
    Runnable r = new Runnable() {
      public void run() {
        proceed(t_min, t_max, t_step );
      }
    };
    threads[i] = new Thread(r);
  }
  for (int i = 1; i < numThreads; i++) {
    threads[i].start();
  }
  threads[0].run();
  try {
    for(int i = 1; i < numThreads; i++) {
      threads[i].join();
    }
  } catch (InterruptedException e) { }
}
```

Figure 5.2 Example of AspectJ loop pointcut (from [HG06])

The example given in *Figure 5.2* shows an application of the `loop()` pointcut, namely parallelization of loops. The example advice executes in parallel (using cyclic loop scheduling) all the loops which are recognized as iterating over a range of integers.

This model of a loop join point presents only an outside view of the loop; the points before and after the loop are not within the loop itself. For some applications it might be desirable to advise the loop body. Also, the loop iterators are good candidates to be advised. AspectMatlab provides a range of poincuts for loops: loop, loopbody and loophead.

5.2 AspectCobol

AspectCobol [LDS05] is inspired from AspectJ in many ways, but it incorporates original techniques for join point identification and context capture. AspectCobol's design strongly suggests that join point reflection on the join point shadow should be viewed as part of the pointcut as opposed to using reflection in the advice code. AspectCobol allows one to extract such details from the join point. The extraction is described as part of the pointcut designator, while the results are bound to variables for subsequent use in the advice code. Hence data is extracted from the shadow of the join point, i.e., the static program context that belongs to the join point.

For example, in *Figure 5.3* we show an aspect that determines an error condition at the time of accessing a file's record, even though Cobol's runtime system does not report any error whatsoever. i.e., any read access to a file's record is to be guarded by a test for the FILESTATUS field to be equal to ZERO (meaning no unhandled error occurred previously). Notice the several bindings of the context information to local variables just before the advice.

While agreeing with the basic approach of AspectCobol, AspectMatlab makes the extraction of context available only at the advice definition level. It enhances the clarity and structure of the whole aspect and also it makes more sense to inquire only the required context information from the static shadow of the join point, right where it is being utilized.

5.3 AOP in MATLAB

There has been some effort made to introduce aspect-oriented features in MATLAB. Joao M. P. Cardoso, et al. [JMPCM06] suggest various useful AOP features, especially those to specify different numeric data types. They have also pointed out the importance of AOP for MATLAB and their work suggests some further use cases. However, our approach includes both general-purpose aspects and specific patterns for scientific applications, as well as a complete and extensible language specification and open-source compiler, including

```
IDENTIFICATION DIVISION.
 ASPECT-ID. ASPECTS/UNSAFEREAD.
DATA DIVISION.
 WORKING-STORAGE SECTION.
 COPY "BOOKS/PANIC.DD".
PROCEDURE DIVISION.
 DECLARATIVES.
  USE BEFORE ANY STATEMENT
  AND BIND VAR-ITEM TO SENDER
  AND VAR-ITEM IS FILE-DATA
  AND BIND VAR-FILE TO FILE OF VAR-ITEM
  AND BIND VAR-STATUS TO FILE-STATUS OF VAR-FILE
  AND BIND VAR-NAME TO NAME OF VAR-FILE
  AND BIND VAR-LOC TO LOCATION
  AND EXISTS PROCEDURE PANIC-STOP
  AND EXISTS DATA PANIC-FIELD.

  MY-UNSAFEREAD-ADVICE.
  IF VAR-STATUS NOT = ZERO
   INITIALIZE PANIC-FIELD
   MOVE VAR-NAME TO PANIC-RESOURCE
   MOVE "UNSAFE READ" TO PANIC-CATEGORY
   MOVE VAR-LOC TO PANIC-LOCATION
   MOVE VAR-STATUS TO PANIC-CODE
   GO TO PANIC-STOP.
 END DECLARATIVES.
```

Figure 5.3 Example of AspectCobol (from [LDS05])

analyses for the dynamic properties of MATLAB.

5.4 Summary

In this chapter, we presented a number of aspect-oriented systems and few extensions to them. While giving an overview of our inspirations from the existing systems, we also discussed the contrast with the approach taken in AspectMatlab.

Although AspectMatlab carries on the basic idea of the pointcuts and the advice from AspectJ, it also introduces a more generic and powerful set of patterns (pointcuts) and the simplified actions (advices). AspectMatlab enables a scientific programmer to cross-cut

the basic program constructs like functions/scripts, arrays and loops.

AspectMatlab provides the facility of selective matching certain join points based on both its actual location in the source code (by using `within` pattern to restrict the scope), and also depending upon the syntax of the join point (by specifying number of arguments used).

Following the AspectCobol design, all necessary context information is only extracted at the actual action level and simply bound to an action's local variables. As compared to the approach taken in AspectJ, where a programmer has to use multiple other pointcuts to fetch the context details, AspectMatlab's approach enhances the clarity.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this section we discuss the contributions made by this thesis. We start with an overview of the driving principles in the designing of the new aspect-oriented scientific language, AspectMatlab. Then we discuss the AspectMatlab compiler (*amc*) and its different phases. We conclude with a brief description of the scientific use cases.

The Design of AspectMatlab is inspired by some motivating factors, such as the introduction of the cross-cutting features for those language constructs, which were not included in the original definitions of the other aspect-oriented languages. Such language constructs include special patterns for arrays, multi-dimensional arrays and loops. Having the ability to cross-cut these language constructs is of utmost importance for a scientific programming language. On top of that, AspectMatlab provides the other patterns (pointcuts) related to the calling and execution of the functions and scripts. The set of patterns is completed with a special scope-restricting pattern used to limit the matching process within a specified program construct.

Another motivating factor behind the design of AspectMatlab was to introduce a simpler, yet extensive, design for the actions (advives). AspectMatlab eliminates the need for pat-

terns to extract the context information from the static shadow of the join point, rather the required context information is defined at the action-definition level and later bound to the action's local variable. AspectMatlab enables a user to inquire about an extensive set of information from the shadow of the join point in the source code. The default ordering rules for actions are simpler and less restrictive than for other aspect-oriented languages. So the accessibility factor for existing and new MATLAB programmers was kept in mind, while designing AspectMatlab.

We have also designed and implemented the AspectMatlab compiler (*amc*) compiler for the new language. The *amc* compiler is designed to be easily extensible, so that other researchers can easily experiment with other new features useful for scientists. The compiler is a source-to-source compiler, producing ordinary MATLAB as its output. This means that any MATLAB system can be used to execute the woven code. The compiler is freely available online at our website[1]. Example programs and the generated code for them are also available.

The *amc* compiler consists of several phases after the front-end in order to make the matching and weaving process accurate. The expression simplification phase converts a complex MATLAB expression into a number of easy to weave statements, only if the sub-expressions might match to the patterns specified in the aspect. All the loop statements in the source code are transformed, again for the sake of creating easy to weave statements.

AspectMatlab presents some challenges for producing correct and efficient woven code. We have described our approach to weaving, including our approach to `around` advice, and the use of a static flow analyses that enabled us to reduce the number of dynamic checks required in the woven code.

We have provided some example use cases that we think indicate the potential for an aspect-oriented system for a scientific programming language. In these examples, the aspects can perform profiling on program features, or they can attribute existing functionality. For example, an aspect can track the growing size of the arrays and report the source code line number of the operations which increase the size. Another aspect can track the sparsity of

[1] www.sable.mcgill.ca/mclab/aspectmatlab

the arrays and can be helpful in making the programs efficient. An aspect can profile the number of floating point operations during the execution of the program. With regard to attributing aspects, an aspect can associate SI units to the variables. Another aspect can interpret the loop iteration space within the loop.

6.2 Future Work

In this section we look into possible improvements to AspectMatlab which would address some of the more important functionality and performance issues with our current implementation. AspectMatlab has the scope to be further evaluated and the performance and functionality can be enhanced.

As far as the extension to the language itself is concerned, different kinds of patterns can be added on top of the existing system. These patterns can target different language constructs, for example, range expressions, try/catch, etc. Several types of operations can be cross-cut as well, for example, array copy operations, arithmetic operations, etc.

Currently in AspectMatlab, actions only obey the precedence rules with respect to the order in which the parent aspect file was presented to the *amc* compiler. The precedence of one specific aspect with respect to the other aspects could be specified as a future extension.

Performance improvement is also part of our future work. Currently, all the aspects are transformed into MATLAB classes, and all the actions calls are actually made to the methods in the class objects. It proves to be an overhead when several calls are made. As compared to Java, there is no code in-lining provided by MATLAB. We believe that the performance of the woven code can be improved using an efficient code in-lining, as it is the case with AspectJ. Improving the performance is vital, since most of the patterns target a lot of static shadows, for example, all the array assignments, or all array accesses, or all the function calls, etc., resulting in frequent aspect class method calls and a performance slowdown.

Another factor which plays an important role in the performance overhead is the MATLAB's semantics for array copying. The context information from the static shadow of the join point is passed on to the actions, and the actual objects involved as a copy. A copy

elimination analysis can further enhance the performance of the aspect woven code.

It is our expectation that scientists will have new and different uses for aspect-oriented programming. In addition to the example use cases we provided, we hope that others will continue to use the language and find new uses and new language extensions.

Appendix A

AspectMatlab Grammar

In *Chapter 2*, the grammar rules for all the concepts and constructs in AspectMatlab language are outlined in pieces. Here we provide a complete definition of the AspectMatlab language definition. If you have a coloured version of this document, you will see that all references to productions in the McLab implementaion of the base MATLAB grammar, are given in red.

⟨program⟩ ::⇒⟨script⟩ | ⟨function_list⟩ | ⟨class⟩ | ⟨aspect⟩
⟨aspect⟩ ::⇒'aspect' IDENTIFIER ⟨stmt_separator⟩ ⟨help_comment⟩
    ⟨aspect_body⟩* 'end'
⟨aspect_body⟩ ::⇒
    ⟨properties_block⟩ ⟨stmt_separator⟩
    | ⟨patterns_block⟩ ⟨stmt_separator⟩
    | ⟨methods_block⟩ ⟨stmt_separator⟩
    | ⟨actions_block⟩ ⟨stmt_separator⟩
⟨patterns_block⟩ ::⇒'patterns' ⟨stmt_separator⟩ ⟨patterns_body⟩* 'end'
⟨patterns_body⟩ ::⇒IDENTIFIER ':' ⟨pattern_designators⟩ ⟨stmt_separator⟩
⟨pattern_designators⟩ ::⇒
    ⟨pattern_designators_and⟩
    | ⟨pattern_designators⟩ '|' ⟨pattern_designators_and⟩

⟨pattern_designators_and⟩ ::⇒
    ⟨pattern_designators_unary⟩
    | ⟨pattern_designators_and⟩ '&' ⟨pattern_designators_unary⟩
⟨pattern_designators_unary⟩ ::⇒
    ⟨pattern_designator⟩
    | '∼' ⟨pattern_designator⟩
⟨pattern_designator⟩ ::⇒
    '(' ⟨pattern_designators⟩ ')'
    | 'set' '(' ⟨pattern_select⟩ ')'
    | 'get' '(' ⟨pattern_select⟩ ')'
    | 'call' '(' ⟨pattern_select⟩ ')'
    | 'execution' '(' ⟨pattern_select⟩ ')'
    | 'mainexecution' '(' ')'
    | 'loop' '(' ⟨pattern_select⟩ ')'
    | 'loopbody' '(' ⟨pattern_select⟩ ')'
    | 'loophead' '(' ⟨pattern_select⟩ ')'
    | 'within' '(' ⟨construct_type⟩ ','⟨pattern_select⟩ ')'
    | IDENTIFIER
⟨pattern_select⟩ ::⇒
    ⟨pattern_target⟩
    | ⟨pattern_target⟩ '(' ⟨list_dotdot⟩ ')'
⟨pattern_target⟩ ::⇒
    ⟨pattern_target_unit⟩
    | ⟨pattern_target⟩ ⟨pattern_target_unit⟩
⟨pattern_target_unit⟩ ::⇒'*' | IDENTIFIER
⟨list_dotdot⟩ ::⇒ε | '..'
    | ⟨list_star⟩
    | ⟨list_star⟩ ',' '..'
⟨list_star⟩ ::⇒'*' | ⟨list_star⟩ ',' '*'
⟨construct_type⟩ ::⇒'*' | 'function' | 'script' | 'loops'
    | 'class' | 'aspect'

⟨actions_block⟩ ::⇒'actions' ⟨stmt_separator⟩ ⟨actions_body⟩* 'end'

⟨actions_body⟩ ::⇒

IDENTIFIER ':' ⟨action_type⟩ IDENTIFIER ⟨stmt_separator⟩
⟨help_comment⟩ ⟨stmt_or_function⟩ 'end'
| IDENTIFIER ':' ⟨action_type⟩ IDENTIFIER ':' ⟨input_params⟩
⟨stmt_separator⟩ ⟨help_comment⟩ ⟨stmt_or_function⟩ 'end'

⟨action_type⟩ ::⇒'before' | 'after' | 'around'

Appendix B

User Manual

A beta-release of the AspectMatlab Compiler (*amc*) is freely available to download from `www.sable.mcgill.ca/mclab/aspectmatlab`.

Once you have a copy of amc.jar, you can execute the jar directly with a list of standard MATLAB files along with AspectMatlab aspect files.

For example, one might run

```
java -jar amc.jar myFunc.m myAspect.m
```

The woven code generated by *amc* can be found in `weaved` directory in the current working directory, which can be executed by any MATLAB system.

B.1 Flags

amc supports the following list of flags:

- A non-aspect MATLAB file can be specified as a starting point of execution with the help of a `-main` flag. For example, `myFunc1.m` is nominated as the entry point.

  ```
  java -jar amc.jar -main myFunc1.m myFunc2.m myAspect.m
  ```

- If standard MATLAB code needs to be translated into Natlab compatible code, use `-m` flag. In the example below, all files following `-m` flag will be translated first.
  `java -jar amc.jar -m myFunc.m myAspect.m`

- An output directory other than the default one can be specified using a `-out` flag. For example:
  `java -jar amc.jar -out output myFunc.m myAspect.m`

- The version number of the *amc* can be checked using a `-version` flag. For example:
  `java -jar amc.jar -version myFunc.m myAspect.m`

- The usage of the *amc* can be checked using a `-h` or a a `-help` flag. For example:
  `java -jar amc.jar -h`
  `java -jar amc.jar -help`

Appendix C

Directory Structure

The AspectMatlab project source has the following directory structure:

- `metaLexer` - AspectMatlab scanner definition files, which are actually an extension to the base Natlab scanner. This directory is an input to the MetaLexer tool.
  - `aspect.mlc` : MetaLexer component for an Aspect
  - `aspect_action.mlc` : MetaLexer component for an Action
  - `aspect_pattern.mlc` : MetaLexer component for a Pattern
  - `aspects_base.mlc` : MetaLexer component which extends the Natlab base component
  - `aspects_matlab.mll` : MetaLexer language file for the AspectMatlab
  - `aspects_start.mlc` : MetaLexer component for the aspect starting state
- `parser` - AspectMatlab parser definition files, which are actually an extension to the base Natlab parser. This directory is an input to the JastAddParser tool.
  - `aspects.parser` : AspectMatlab grammar definition
  - `header.parser` : Header file to specify package name and imports

- `jastadd` - AspectMatlab abstract syntax tree (AST) and a collection of attribute files. These attribute files encode the several integral components of the AspectMatlab compiler at the AST node level. They contain the functionality for weeding the source code, expression simplification, loop transformation, context information, matching and weaving, and finally pretty printing the woven code. This directory is an input to the JastAdd tool.

  - `aspects.ast` : AspectMatlab abstract syntax tree
  - `AspectsCorrespondingFunctions.jrag` : JastAdd attribute for expression simplification phase
  - `AspectsInheritedEquations.jrag` : JastAdd attribute to specify equations for the attributes inherited from Natlab
  - `AspectWeave.jrag` : JastAdd attribute for weaving phase
  - `AssignStmtWeavability.jadd` : JastAdd attribute to determine the weaveabilty of an assignment statement (used to distinguish the statements inserted by the compiler)
  - `ContextInfo.jadd` : JastAdd attribute to keep the line number for each statement node
  - `FetchTargetExpr.jrag` : JastAdd attribute to determine the target variable within an expression
  - `FileName.jadd` : JastAdd attribute to keep the file names
  - `GlobalStructure.jrag` : JastAdd attribute for the post-processing phase
  - `LoopTransformation.jadd` : JastAdd attribute for loop transformation phase
  - `PrettyPrint.jrag` : JastAdd attribute to print out the standard MATLAB code
  - `ProceedTransformation.jrag` : JastAdd attribute to translate the `proceed` calls within an `around` action

- `ShadowMatch.jrag` : JastAdd attribute for matching phase
- `WeaveLoopStmts.jrag` : JastAdd attribute for inserting the action calls inside a loop's body at certain points with respect to `break`, `continue`, and `return` statements
- `Weeding.jadd` : JastAdd attribute for weeding phase

• `aspectMatlab` - AspectMatlab source Java files, which includes the program entry point and the complete AspectMatlab system.

- `ActionInfo.java` : Class definition for the `ActionInfo` structure used in `AspectInfo`
- `AspectEngine.java` : Class definition for the complete AspectMatlab functionality
- `Main.java` : AspectMatlab entry point

Appendix D

Scientific Aspects

D.1 Tracking operations that grow arrays

```
aspect grow
   % this aspect catches every set and records data that should be
   % useful in determining which operations increase or decrease the
   % array size. to that effect the size of every variable during the
   % run of the program is checked. In the end, the line number of the
   % operation at which the size of each array was maximum, is printed
   % out along with the size.

 properties
   variables = struct(); % creates the mapping 'variable' -> index
   changeShape = {}; % how often the dimensions of the array changed
      (has to exist previously)
   decreaseSize = {}; % how often the size decreased (i.e. a previously
      nonzero element was set)
   increaseSize = {}; % how often the size increased
   arraySize = {}; % size of the array
   maxSize = {}; % maximum size of the array
   lineNum = {}; % at line number
   arraySet = {}; % the number of 'set' operations
```

```
  nextId = 1; % next available index
end

methods
  function b = sameShape(this,a,b)
    % returns true if a and b have the same shape
    if (ndims(a) ~= ndims(b))
      b = false;
    elseif (size(a) == size(b))
      b = true;
    else
      b = false;
    end
  end

  function id = getVarId(this,var,line)
    % get id of variable by string-name, update 'variables' if
        necessary
      if (~isfield(this.variables,var))
        this.variables = setfield(this.variables,var,this.nextId);
        id = this.nextId;
        this.nextId = this.nextId+1;
        % initialze entry <id> for all the cell arays
        this.arraySet {id} = 0; % the number of 'set' operations
        this.changeShape{id} = 0; % how often the dimensions of the
            array changed (has to exist previously)
        this.decreaseSize{id} = 0; % how often the size decreased (i.e.
            a previously nonzero element was set)
        this.increaseSize{id} = 0; % how often the size increased
        this.arraySize{id} = 0;
        this.maxSize{id} = 0;
        this.lineNum{id} = line;
      else
        id = getfield(this.variables,var);
      end
  end
end
```

```
patterns
  arraySet : set(*);
  exec : execution(program);
end

actions
  message : before exec
    disp('tracking the operations that grow arrays in the following
        program...');
  end

  displayResults : after exec
    % will display the results
      vars = fieldnames(this.variables);
      result = {'var','arraySet','shape
          changes','decrease','increase','max size','line#'};
      pm = [' ', char(0177)];
      for i=1:length(vars) %iterate over variables
        result{i+1,1} = vars{i};
        result{i+1,2} = this.arraySet{i};
        result{i+1,3} = this.changeShape{i};
        result{i+1,4} = this.decreaseSize{i};
        result{i+1,5} = this.increaseSize{i};
        result{i+1,6} = this.maxSize{i};
        result{i+1,7} = this.lineNum{i};
      end
      disp(result);
  end

  set : before arraySet : (newVal,obj,name,line,args)
      t = obj;
      t(args{1:numel(args)}) = newVal;
      newVal = t;

      % we will exit if the newval is not a matrix
      if (~isnumeric(newVal))
```

```
      return;
    end;

    % get id of variable by string-name, update 'variables' if
        necessary
    id = this.getVarId(name,line);

    % get var infor
    newSize = numel(newVal);
    oldSize = this.arraySize{id};

    this.arraySize{id} = newSize;

    % update the number of 'set' operations
    this.arraySet{id} = this.arraySet{id}+1;

    % set shape/sparsity changes
    if (~this.sameShape(newVal,obj))
      % how often the dimensions of the array changed (has to exist
          previously)
      this.changeShape{id} = this.changeShape{id}+1;
    end
    if (newSize < oldSize)
      % how often the size decreased
      this.decreaseSize{id} = this.decreaseSize{id}+1;
    end;
    if (newSize > oldSize)
      % how often the size increased
      this.increaseSize{id} = this.increaseSize{id}+1;
      this.lineNum{id} = line;
      this.maxSize{id} = newSize;
    end
  end
 end
end
```

D.2 Tracking array sparsity

```
aspect sparsity
  % this aspect catches every set and records data that should be
  % useful in determining which variables can safely be declared
  % as sparse. to that effect the sparsity of every variable during
  % the run of the program is checked, as well as how often the size
  % of the array and the sparsity changes. the standard deviation on
  % the sparsity is checked as well. also tracks sizes of variables
  % (and stdev). these values are tracked for all variables over the
  % run of the whole program, for all sets and gets

 properties
  variables = struct(); % creates the mapping 'variable' -> index
  sizeSum = {}; % the sum of size of variables
  sizeSumSquared = {}; % the sum size of variables squared - to
     calculate stdev
  sparsitySum = {}; % the sum of sparsity
  sparsitySumSquared = {};% the sum of the sparsity squared - to
     calculate stdev
  changeShape = {}; % how often the dimensions of the array changed
     (has to exist previously)
  decreaseSparsity = {}; % how often the sparsity decreased (i.e. a
     previously nonzero element was set)
  increaseSparsity = {}; % how often the sparsity increased
  arraySet = {}; % the number of 'set' operations
  arrayGet = {}; % how often the whole array is retrieved
  arrayIndexedGet = {}; % how often the array is indexed into
  nextId = 1; % next available index
 end

 methods
  function b = sameShape(this,a,b)
   % returns true if a and b have the same shape
   if (ndims(a) ~= ndims(b))
    b = false;
```

```
  elseif (size(a) == size(b))
    b = true;
  else
    b = false;
  end
end

function s = stdev(this,sum,sumSquared,N)
  mean = sum/N;
  s = sqrt(sumSquared/N - mean^2);
  if (sumSquared/N < mean^2) % make numerical errors not report
      imaginary results
    s = 0;
  end;
end

function id = getVarId(this,var)
  % get id of variable by string-name, update 'variables' if
      necessary
    if (~isfield(this.variables,var))
      this.variables = setfield(this.variables,var,this.nextId);
      id = this.nextId;
      this.nextId = this.nextId+1;
      % initialze entry <id> for all the cell arays
      this.sizeSum{id} = 0; % the sum of size of variables
      this.sizeSumSquared{id} = 0; % the sum size of variables
          squared - to calculate stdev
      this.sparsitySum{id} = 0; % the sum of sparsity
      this.sparsitySumSquared{id} = 0;% the sum of the sparsity
          squared - to calculate stdev
      this.arraySet {id} = 0; % the number of 'set' operations
      this.changeShape{id} = 0; % how often the dimensions of the
          array changed (has to exist previously)
      this.decreaseSparsity{id} = 0; % how often the sparsity
          decreased (i.e. a previously nonzero element was set)
      this.increaseSparsity{id} = 0; % how often the sparsity
          increased
```

```
      this.arrayGet{id} = 0; % how often the whole array is retrieved
      this.arrayIndexedGet{id} = 0; % how often the array is indexed
          into
    else
      id = getfield(this.variables,var);
    end
  end

  % returns sparsity
  function s = getSparsity(this,val)
    if (numel(val) == 0)
      s = 1;
    else
      s = nnz(val)/numel(val);
    end
  end

  % given some matrix and a var id, updates the size, sparsity fields
  function s = touch(this,id,value)
    sp = this.getSparsity(value);
    newSize = numel(value);
    this.sizeSum{id} = this.sizeSum{id}+newSize; % add new size
    this.sizeSumSquared{id} = this.sizeSumSquared{id}+newSize^2; % add
        new size squared
    this.sparsitySum{id} = this.sparsitySum{id}+sp; % add to the sum
        of sparsity
    this.sparsitySumSquared{id} = this.sparsitySumSquared{id}+sp^2; %
        add sum squared
  end
end

patterns
  arraySet : set(*);
  arrayWholeGet : get(*());
  arrayIndexedGet : get(*(..));
  exec : execution(program);
end
```

```
actions
  message : before exec
   disp('tracking sparsities of all variables in the following
       program...');
  end

  displayResults : after exec
  % will display the results
    vars = fieldnames(this.variables);
    result = {'var','size','sparsity','arraySet','shape
        changes','decrease sparsity','increase sparsity','get','indexed
        get'};
    pm = [' ', char(0177)];
    for i=1:length(vars) %iterate over variables
      result{i+1,1} = vars{i};
      touch =
          this.arraySet{i}+this.arrayGet{i}+this.arrayIndexedGet{i}; %
          total number of acesses
      result{i+1,2} =
          strcat(num2str(this.sizeSum{i}/touch,'\%.1f'),pm,num2str(this.stdev(
           this.sizeSum{i},this.sizeSumSquared{i},touch),'\%.1f'));
      result{i+1,3} =
          strcat(num2str(this.sparsitySum{i}/touch,'\%1.2f'),pm,num2str(this.stdev(
           this.sparsitySum{i},this.sparsitySumSquared{i},touch),'\%1.2f'));
      result{i+1,4} = this.arraySet{i};
      result{i+1,5} = this.changeShape{i};
      result{i+1,6} = this.decreaseSparsity{i};
      result{i+1,7} = this.increaseSparsity{i};
      result{i+1,8} = this.arrayGet{i};
      result{i+1,9} = this.arrayIndexedGet{i};
    end
    disp(result);
  end

  set : before arraySet : (newVal,obj,name,args)
     t = obj;
```

```
 t(args{1:numel(args)}) = newVal;
 newVal = t;

% we will exit if the newval is not a matrix
if (~isnumeric(newVal))
  return;
end;

% get id of variable by string-name, update 'variables' if
    necessary
id = this.getVarId(name);

% get var infor
newSize = numel(newVal);
sparsity = this.getSparsity(newVal);
oldSparsity = this.getSparsity(obj);

% update the number of 'set' operations
this.arraySet{id} = this.arraySet{id}+1;

% update tracking info
this.touch(id,newVal);

% set shape/sparsity changes
if (~this.sameShape(newVal,obj))
  % how often the dimensions of the array changed (has to exist
      previously)
  this.changeShape{id} = this.changeShape{id}+1;
end
if (sparsity < oldSparsity)
  % how often the sparsity decreased
  this.decreaseSparsity{id} = this.decreaseSparsity{id}+1;
end;
if (sparsity > oldSparsity)
  % how often the sparsity increased
  this.increaseSparsity{id} = this.increaseSparsity{id}+1;
end
```

```
    end

    get : before arrayWholeGet : (obj,name)
      % we will exit if the value is not a matrix
      if (~isnumeric(obj))
        return;
      end;
      id = this.getVarId(name); % get id of variable by string-name,
          update 'variables' if necessary
      this.touch(id,obj);
      this.arrayGet{id} = this.arrayGet{id}+1;
    end

    indexedGet : before arrayIndexedGet : (obj,name)
      % we will exit if the value is not a matrix
      if (~isnumeric(obj))
        return;
      end;
      id = this.getVarId(name); % get id of variable by string-name,
          update 'variables' if necessary
      this.touch(id,obj);
      this.arrayIndexedGet{id} = this.arrayIndexedGet{id}+1;
    end
  end
end
```

D.3 Measuring floating point operations

```
aspect flops
  % catches
  % mul, plus, minus, mtimes, time, plus, sqrt, rdivide, abs
  %
  % and records the number of flops
  % - in total
  % - for every call
```

```
% - the number of calls
% per call site, and records the data recursively
%
% uses a stack
% before any call, creates a new 'stack frame' with number of flops
   of operation
% after any call, destroys stack frame, puts the flops of that
   stackframe
% on the new top, and updates call site info
%
% for builtin functions we use an around that adds the flops to the
   top
% of the stack, with a proceed
%
% this aspect gives detailed flops infor for every call of 'SVD'
% but that behaviour can be overriden by simply changing the
   'tracking' pointcut

properties
  callSite = struct(); % callsite -> id
  call = []; % number of calls per call site
  flop = []; % flops per call site
  nextId = 1;
  s = [1, 0]; % put sth in stack=> calls can modify the 'top' without
     error
  record = false;
end

methods
  % stack methods - stack(1) is the number of elements, which itself
     follow
  function s=stack(this)
    s=[0];
  end
  function stack=push(this,stack,element)
    stack(stack(1)+2) = element;
    stack(1) = stack(1)+1;
```

```
  end
  function [stack,element]=pop(this,stack)
    if (stack(1) == 0)
      error('trying to pop from empty stack');
    end
    element = stack(stack(1)+1);
    stack(1) = stack(1)-1;
  end

  % given a scope (function) name (location), line number and
      operation, returns the
  % associated index in calls and flops
  function id = getId(this,name,line,op)
    location = strcat(name,'_',num2str(line),'_',op);
    if (~isfield(this.callSite,location))
      this.callSite=setfield(this.callSite,location,this.nextId);
      this.flop(this.nextId) = 0;
      this.call(this.nextId) = 0;
      id = this.nextId;
      this.nextId = this.nextId + 1;
    else
      id = getfield(this.callSite,location);
    end
  end
end

patterns
  tracking: call(SVD);

  pminus : call(minus (*,*));
  pmtimes : call(mtimes (*,*));
  ptimes : call(times (*,*));
  pplus : call(plus (*,*));
  psqrt : call(sqrt (*));
  prdivide: call(rdivide(*,*));
  pabs : call(abs (*));
```

```
  any : call(*);
end

actions
  % before tracked call set up vars
  beforeTrack : before tracking : (name)
    fprintf('encountered call to %s, recording flops...\n',name);
    this.callSite = struct(); % callsite -> id
    this.call = []; % number of calls per call site
    this.flop = []; % flops per call site
    this.nextId = 1;
    this.s = this.stack();
    this.record = true;
  end

  % before any call - take care of loops on stack (if recording)
  % this gets called after the beforeTrack advice, so that the tracked
      call can
  % report information
  bany : before any
    if (~this.record)
      return; % return if we are not recording
    end
    this.s = this.push(this.s,0);
    end

  % after call - store info and put flops on previous 'stack frame'
  % 'aany' should get called first, because a call to the tracking
      function
  % should still list said call with the corresponding flops
      information
  aany : after any : (name,line,loc);
    if (~this.record)
      return; % return if we are not recording
    end
    [this.s,f] = this.pop(this.s); % get flops and return stack
    id = this.getId(loc,line,name);
```

```
  this.call(id) = this.call(id) + 1;
  this.flop(id) = this.flop(id) + f;
  % if the stack isn't empty, put all those flops on the previous
      frame
      if (this.s(1) ~= 0)
       [this.s, fold] = this.pop(this.s);
       this.s = this.push(this.s,f + fold);
  end
end

% after tracked call print out results
afterTrack : after tracking
  % print info
  fprintf('finished tracking function call, here are the
      results:\n');
  fields = fieldnames((this.callSite));
  result = {'call site','# of calls','total flops'};
  format('long');
  for i = 1:numel(fields);
    field = fields{i};
    id = getfield(this.callSite, field);
    result{i+1,1} = field;
    result{i+1,2} = this.call(id);
    result{i+1,3} = this.flop(id);
  end
  disp(result);
  % put something in the stack so that calls can modify the 'top'
      without error
  this.s = this.push(this.stack(),0);
  this.record = false;
end

%the operations
% we assume matrix multiplication of A:mxn, B:nxk takes (2n-1)*k*m
    operations
amtimes : around pmtimes : (args)
  proceed(); % first perform call
```

```
  f = (2*size(args{1},2) - 1)*size(args{1},1)*size(args{2},2);
  [this.s,fold] = this.pop(this.s);
  this.s = this.push(this.s,f+fold);
end

% binary element-wise operations
aminus : around pminus : (args)
  proceed(); % first perform call
  f = max(numel(args{1}),numel(args{2}));
  [this.s,fold] = this.pop(this.s);
  this.s = this.push(this.s,f+fold);
end
atimes : around ptimes : (args)
  proceed(); % first perform call
  f = max(numel(args{1}),numel(args{2}));
  [this.s,fold] = this.pop(this.s);
  this.s = this.push(this.s,f+fold);
end
aplus : around pplus : (args)
  proceed(); % first perform call
  f = max(numel(args{1}),numel(args{2}));
  [this.s,fold] = this.pop(this.s);
  this.s = this.push(this.s,f+fold);
end
ardivide : around prdivide : (args)
  proceed(); % first perform call
  f = max(numel(args{1}),numel(args{2}));
  [this.s,fold] = this.pop(this.s);
  this.s = this.push(this.s,f+fold);
end

% unary element wise operations
asqrt : around psqrt : (args)
  proceed(); % first perform call
  f = (numel(args{1}));
  [this.s,fold] = this.pop(this.s);
  this.s = this.push(this.s,f+fold);
```

```
    end
    aabs : around pabs : (args)
      proceed(); % first perform call
      f = (numel(args{1}));
      [this.s,fold] = this.pop(this.s);
      this.s = this.push(this.s,f+fold);
    end
  end
end
```

D.4 Adding units to computations

```
aspect unit
    % allows adding units
    % units are SI and SI derived units
    % the unit is denoted by a vector
    % ====[metre, kg, second, Ampere, Kelvin, candela, mol]======
    % all acesses to functions denoted by units are overriden
    % all operations are overriden
    % indexing gets overriden
    % uses structs using the aspect_annoted flag

  properties
    noUnit = [0, 0, 0, 0, 0, 0, 0];
    annotated = 'aspect_annotated';
    one = struct('aspect_annotated',true,'val',1,'unit',[0, 0, 0, 0, 0,
        0, 0]);
    units = struct(... % defines all SI and SI derived unit names and
        value (may be used for printing as well)
      'm', [1, 0, 0, 0, 0, 0, 0],...
      'Kg', [0, 1, 0, 0, 0, 0, 0],...
      's', [0, 0, 1, 0, 0, 0, 0],...
      'A', [0, 0, 0, 1, 0, 0, 0],...
      'K', [0, 0, 0, 0, 1, 0, 0],...
      'cd', [0, 0, 0, 0, 0, 1, 0],...
```

```
    'mol',[0, 0, 0, 0, 0, 0, 1],...
    'J', [2, 1,-2, 0, 0, 0, 0],...
    'N', [1, 1,-2, 0, 0, 0, 0]);
  constants = struct(... % defines constants or units whose factor
      (compared to SI units) are not 1
    'km', {[1, 0, 0, 0, 0, 0, 0],1000},...
    'year', {[0, 0, 1, 0, 0, 0, 0],31556926},...
    'lb', {[0, 1, 0, 0, 0, 0, 0],0.45359237},...
    'inches', {[1, 0, 0, 0, 0, 0, 0],0.0254},...
    'feet', {[1, 0, 0, 0, 0, 0, 0],0.3048},...
    'G', {[3,-1,-2, 0, 0, 0, 0], 6.6730e-11},...
    'dozen', {[0, 0, 0, 0, 0, 0, 0],12},...
    'AU', {[1, 0, 0, 0, 0, 0, 0],149598000*1000},...
    'c', {[1, 0,-1, 0, 0, 0, 0],299792458},...
    'KJ', {[2, 1,-2, 0, 0, 0, 0],1000},...
    'g', {[0, 1, 0, 0, 0, 0, 0],0.001},...
    'L', {[3, 0, 0, 0, 0, 0, 0],0.001},...
    'kilotons',{[0, 1, 0, 0, 0, 0, 0],1000*1000},...
    'm_earth', {[0, 1, 0, 0, 0, 0, 0],5.9742e24},...
    'r_earth', {[1, 0, 0, 0, 0, 0, 0],6378100});
end

methods
  function s = annotate(this,x)
  % takes in a value and returns value that is unit-annotated for sure
  % if it is annotated already, the same unit is returned
    if (isstruct(x) && isfield(x,this.annotated))
      s = x;
    else
      s = struct(this.annotated,true,'val',x,'unit',this.noUnit);
    end
  end

  function [a,b,c] = prepareOp(this,args)
  % prepares input args a,b and output arg c for binary operation --
      this
  % is just common code put in a separate function
```

```
  if (length(args) ~= 2)
    error(strcat('binary operation needs exactly 2 arguments'));
  end
  a = this.annotate(args{1});
  b = this.annotate(args{2});
  c = this.one;
end

% displays a unit on screen
function display(this,v)
  if ((isstruct(v)) && isfield(v, this.annotated))
    fprintf('%s:',this.unitString(v.unit)); disp(v.val);
  else
    disp(v);
  end
end

function s = unitString(this,v)
% returns the unit string of a given unit vector
% this is done greedily/recursively by picking the unit that most
    reduces the 1-norm
% of the unit vector.
  s = '';
  if (v == this.noUnit)
    return;
  end
  names = fieldnames(this.units);
  print = zeros(length(names), 1);

  % this loop picks the unit that most reduces the 1-norm of v,
  % and adds it to 'print' until v is 0
  while (~same(v,0*v))
    newPNorm = (print * 0);
    newMNorm = (print * 0);
    for i = (1 : length(names))
      u = getfield(this.units, names{i}); % get vector for unit i
```

```
      newPNorm(i) = norm((v - u), 1); % see how much that unit
         reduces the unit vector v
      newMNorm(i) = norm((v + u), 1); % same but with inverted unit
    end
    [minPNorm, minPi] = min(newPNorm);
    [minMNorm, minMi] = min(newMNorm);
    if (minPNorm < minMNorm) % put the found unit into print vector
      print(minPi) = (print(minPi) + 1); % positive unit (unit^1)
      u = -getfield(this.units, names{minPi});
    else
      print(minMi) = (print(minMi) - 1); % negativ unit (unit^-1)
      u = (getfield(this.units, names{minMi}));
    end
    v = (v + u);
  end

  % put whatever is in the print vector into a string
  for i = (1 : length(print))
    if (print(i) ~= 0)
      s = strcat(s, strcat('*', names{i}));
      if (print(i) ~= 1)
        s = strcat(s, strcat('^', num2str(print(i))));
      end
    end
  end
  s = s(2:length(s)); % we know there must be one unit - replace
     leading '*'
 end
end

patterns
 disp : call(disp);
 plus : call(plus(*,*));
 minus : call(minus(*,*));
 mtimes : call(mtimes(*,*));
 mrdivide : call(mrdivide(*,*));
 power : call(power(*,*));
```

```
  round : call(round(*));
  colon : call(colon(*,..));
  allCalls : call(*());
  loopheader : loophead(*);
end

actions
  % captures all loop invocations for i = range, and overwrites the
  % expression to be a struct-array instead of a structure with an
     array inside
  loop : around loopheader : (newVal)
    range = this.annotate(newVal);
    % loop through range.val, and record whatever the for loop
       captures in a cell array
    acell = {};
    for i = (range.val)
      acell{length(acell)+1} = i;
    end
    varargout{1} =
       struct(this.annotated,true,'val',acell,'unit',range.unit);
  end

  acalls : around allCalls : (name)
  % captures all calls and checks whether they are a nuit - if so,
     return the unit
  % this advice is first so that it gets matched last
    if (isfield(this.units,name))
     varargout{1} =
        struct(this.annotated,true,'val',1,'unit',getfield(this.units,name));
    else
     if (isfield(this.constants,name))
      pair = getfield(this.constants,name);
      varargout{1} = struct(this.annotated, true, 'val', getfield(
              this.constants,{2},name), 'unit',
                 getfield(this.constants,{1},name));
     else
      proceed();
```

```
    end
  end
end

adisp : around disp : (args)
% overrdes printing so that we add units
  if (length(args) ~= 1)
    error('Error using disp -- need exactly one argument)');
  end
  v = args{1};
  if (isstruct(v) && isfield(v,this.annotated))
    this.display(v);
  else
    disp(v);
  end
end

aplus : around plus : (args)
% +
  [a,b,c] = this.prepareOp(args);
  c.val = a.val+b.val;
  if (a.unit ~= b.unit)
    error('the units of the arguments for operation + must match');
  end
  c.unit = a.unit;
  varargout{1} = c;
end

aminus : around minus : (args)
% -
  [a,b,c] = this.prepareOp(args);
  c.val = a.val-b.val;
  if (a.unit ~= b.unit)
    error('the units of the arguments for operation - must match');
  end
  c.unit = a.unit;
  varargout{1} = c;
```

```
end

amtimes : around mtimes : (args)
% *
  [a,b,c] = this.prepareOp(args);
  c.val = a.val*b.val;
  c.unit = a.unit+b.unit;
  varargout{1} = c;
end

amrdivide : around mrdivide : (args)
% /
  [a,b,c] = this.prepareOp(args);
  c.val = a.val/b.val;
  c.unit = a.unit-b.unit;
  varargout{1} = c;
end

power : around power : (args)
% .^
  [a,b,c] = this.prepareOp(args);
  c.val = a.val.^b.val;
  if (b.unit ~= this.noUnit)
    error('cannot use power with a non empty unit');
  end
  if (isscalar(b.val))
    c.val = a.val.^b.val;
    c.unit = a.unit*b.val;
  else
    if (a.unit ~= this.noUnit)
      error('cannot use power operation resulting mixed unit matrix');
    end
    c.unit = this.noUnit;
  end
  varargout{1} = c;
end
```

```
    round : around round : (args)
    % round
      if (length(args) ~= 1)
        proceed();
      end
      a = this.annotate(args{1});
      a.val = round(a.val);
      varargout{1} = a;
    end

    colon : around colon : (args)
    % : :
      if (length(args) ~= 2 && length(args) ~= 3)
        proceed();
      end
      a = this.annotate(args{1});
      b = this.annotate(args{2});
      c = this.one;
      o = this.one;
      o.unit = a.unit;
      if (b.unit ~= a.unit)
        error('error in colon: the units need to be the same');
      end
      if (length(args) == 3)
        c = this.annotate(args{3});
        if (c.unit ~= a.unit)
          error('error in colon: the units need to be the same');
        end
        o.val = a.val:b.val:c.val;
      else
        o.val = a.val:b.val;
      end
      varargout{1} = o;
    end
  end
end
```

D.5 Interpreting loop iteration space

```
aspect loops
  properties
     stack = {};
     top = 1;
  end

  methods
    function push(this, s)
       if(numel(s) > 0)
            this.stack{this.top}.lbound = s(1);
            this.stack{this.top}.ubound = s(numel(s));
            this.stack{this.top}.increment = this.increment(s);
         else
            this.stack{this.top}.lbound = NaN;
            this.stack{this.top}.ubound = NaN;
            this.stack{this.top}.increment = NaN;
         end
       this.stack{this.top}.iteration = 0;
       this.top = this.top + 1;
    end

    function pop(this)
       this.top = this.top - 1;
    end

    function lb = getLBound(this)
       lb = this.stack{this.top-1}.lbound;
    end

    function ub = getUBound(this)
       ub = this.stack{this.top-1}.ubound;
    end

    function inc = getIncrement(this)
```

```
      inc = this.stack{this.top-1}.increment;
    end

    function iteration = getIteration(this)
      iteration = this.stack{this.top-1}.iteration;
    end

    function update(this, iteration)
      this.stack{this.top-1}.iteration = iteration;
    end

    function inc = increment(this, s)
      size = numel(s);
      first = s(1);
      last = s(size);
      step = (last-first)/(size-1);
      if(s(1):step:s(size) == s)
        inc = step;
      else
        inc = NaN;
      end
    end
  end

  patterns
    ploophead : loophead(*);
    ploopbody : loopbody(*);
    ploop : loop(*);

    lbound : call(lBound) & within(loops,*);
    ubound : call(uBound) & within(loops,*);
    increment : call(increment) & within(loops,*);
    iteration : call(iteration) & within(loops,*);
  end


  actions
```

```
    aLoopHead : after ploophead : (newVal)
       this.push(newVal);
    end

    aLoopBody : before ploopbody : (counter)
       this.update(counter);
       end

    aLoop : after ploop
       this.pop();
       end

    aLBound : around lbound
       % captures all loop invocations for lBound
       varargout{1} = this.getLBound();
    end

    aUBound : around ubound
       % captures all loop invocations for uBound
       varargout{1} = this.getUBound();
    end

    aIncrement : around increment
       % captures all loop invocations for increment
       varargout{1} = this.getIncrement();
    end

    aIteration : around iteration
       % captures all loop invocations for iteration
       varargout{1} = this.getIteration();
    end
  end
end
```

Bibliography

[abc] abc. The AspectBench Compiler. Home page `http://aspectbench.org`.

[ACH$^+$05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *AOSD*, mar 2005, pages 87–98. ACM Press.

[Asp03] AspectJ Eclipse Home. The AspectJ home page. http://eclipse.org/aspectj/, 2003.

[Cas09] Andrew Casey. The metalexer lexical specification language. Master's thesis, McGill University, September 2009.

[CC07] Kung Chen and Chin-Hung Chien. Extending the field access pointcuts of AspectJ to arrays. *Journal of Software Engineering Studies*, 2(2):93–102, June 2007.

[EH07] Torbjörn Ekman and Görel Hedin. The Jastadd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, Montreal, Quebec, Canada, 2007, pages 1–18. ACM, New York, NY, USA.

[HG06] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, Bonn, Germany, 2006, pages 63–74. ACM, New York, NY, USA.

[JMPCM06] Joo M. Fernandes Joo M. P. Cardoso and Miguel P. Monteiro. Adding aspect-oriented features to MATLAB, March 2006. Proceedings of SPLAT workshop at AOSD '06 `www.aosd.net/workshops/splat/2006/papers/cardoso.pdf`.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, 2001, volume 2072, pages 327–353.

[Kuz04] Sascha Kuzins. Efficient implementation of around-advice for the aspectbench compiler. Master's thesis, Oxford University, September 2004.

[LDS05] Ralf Lämmel and Kris De Schutter. What does aspect-oriented programming mean to Cobol? In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, Chicago, Illinois, 2005, pages 99–110. ACM, New York, NY, USA.

[Mat] Matlab. The Language Of Technical Computing. Home page `http://www.mathworks.com/products/matlab/`.

[RLB05] J. Douglas Faires Richard L. Burden. *Numerical Analysis, eigth edition*. Thomson Books/Cole, Belmont, California, 2005.

[TAH10] Anton Dubrau Toheed Aslam, Jesse Doherty and Laurie Hendren. Aspectmatlab: An aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th international conference on Aspect-oriented software development*, Rennes and St. Malo, France, 2010. ACM, New York, NY, USA.

[Wat02] David S. Watkins. *Fundamentals of Matrix Computations, second edition*. John Wiley and Sons, 2002.