ASPECTMATLAB++: DEVELOPING AN ASPECT-ORIENTED LANGUAGE FOR SCIENTISTS

by

Andrew Bodzay

School of Computer Science
McGill University, Montréal

Sunday, December 7th 2014

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

Abstract

MATLAB is a popular dynamic array-based language commonly used within the scientific community. MATLAB's widespread use can be attributed to its large library of built-in functions, and its high-level syntax, which requires no type declarations, making it ideal for fast prototyping. This thesis presents extensions to ASPECTMATLAB, an aspect oriented compiler developed for MATLAB.

ASPECTMATLAB was created with the intent of bringing aspect oriented programming to MATLAB, and targeted features such as array accesses and loops, which are the core computations in scientific programs. This thesis presents ASPECTMATLAB++. ASPECTMATLAB++ extends ASPECTMATLAB by focusing on a different set of challenges, seeking to make aspect-oriented programming easier to use and providing mechanisms to handle a variety of the problems that occur in a dynamically typed language. To this end, we introduce pattern matching on annotations and types of variables, as well as new manners of exposing context.

We also provide several use-cases of these features in the form of general-use aspects which focus on solving issues that arise from use of dynamically-typed languages. These include aspects which perform type and unit checking, profiling aspects, as well as aspects which perform basic loop optimizations. This thesis also details several performance enhancements to the ASPECTMATLAB compiler, which result in a speed improvement of about 10 times.

Résumé

MATLAB est une langage de programmation dynamique utilisée partout dans la communauté scientifique. L'utilisation répandu de MATLAB peut être attribué à sa grande bibliothèque de fonctions incorporées et sa syntaxe de haut niveau, qui n'exige aucune déclaration de type, le faisant idéal pour prototypage rapide. Cette thèse présente des extensions à ASPECTMATLAB, un compilateur orientée à l'aspect développé pour MATLAB.

ASPECTMATLAB a été créé avec l'intention d'amener programmation orientée à l'aspect à MATLAB. Il cible des fonctionalités comme des accès de tableau et des boucles, qui sont les calculs principaux dans des programmes scientifiques. Cette thèse présente ASPECTMATLAB++. ASPECTMATLAB++ étend ASPECTMATLAB en se concentrant sur un ensemble différent de défis, cherchant à faciliter la programmation orientée à l'aspect et introduire des mécanismes qui gére une variété des problèmes qui arrivent dans une langue dynamiquement tapée. À cette fin, nous présentons correspondance sur des annotations et les types de variables, et des nouvelles manières d'exposer le contexte.

Nous fournissons aussi plusieurs cas d'utilisation de ces fonctions en forme des aspects d'utilisation générale qui se concentrent sur les questions de résolution qui résultent de l'utilisation de langues dynamiquement tapées. Ceux-ci incluent les aspects qui exécutent le type et la vérification d'unité, des aspects de profilage, aussi bien que les aspects qui exécutent des optimisations de boucles. Cette thèse fournit aussi plusieurs augmentations de performance au compilateur ASPECTMATLAB, qui donnent une amélioration de vitesse d'environ 10 fois.

Acknowledgements

First, I would like to extend my gratitude to my supervisor, Professor Laurie Hendren, for providing me with the opportunity to work on this project. Her support, encouragement and guidance were crucial to the success of this work.

In addition, I would like to thank all the members of the *Mc*LAB team for their contributions to the *Mc*LAB project. The *Mc*LAB framework has been an excellent platform with which to work.

Finally, I would like to thank my friends and family for their ongoing support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

MATLAB [Molb, Mola, Mat09] is a dynamic array-based programming language, which has widespread use throughout the scientific community. Its success stems from its convenience as a dynamic scripting language - providing the programmer with high-level matrix operators, a large library of built-in functions, a flexible syntax which requires no type declarations, as well as a quick and easy development through the MATLAB IDE. These factors all make the language very appealing to novice and expert programmers alike, as development requires little training, and code can be rapidly prototyped.

Our work, ASPECTMATLAB++ builds upon the successes of the ASPECTMATLAB project, which introduced the idea of aspects in MATLAB [ADDH10, Asl10]. The original focus of ASPECTMATLAB was that prominent features in scientific programming, such as loops and arrays could be easily matched and operated upon. It was designed to be easily understood by those familiar with the MATLAB programming language, and did this by extending object-oriented MATLAB classes. Aspects have properties and methods, similar to MATLAB classes, but also allow for patterns, which specify sets of join points, as well as actions, code associated with patterns to be executed at join points. Our challenge was to further develop aspect-oriented programming for MATLAB, in a way that is consistent with the ease of use of the base language. We wanted to introduce language mechanisms through which programmers could more easily express their intent and control their code, with the goal of preventing errors as well as improving performance. Simultaneously, we aim to make ASPECTMATLAB more accessible to current MATLAB users.

Unlike more formal programming languages, MATLAB has neither static type declarations nor static type checking, with type information being determined at run-time. Despite the fact that no formal type declarations exist, many MATLAB functions have comments which specify the types expected of its arguments. Failure to meet these informal recommendations can result in not only run-time errors, but also incorrect results. So while forgoing static typing does make prototyping quick, it can also result in type inconsistencies which can propagate throughout a program. One of our aims is to accommodate scientists by creating language extensions and scientific aspects which help them to understand and deal with these sorts of typing issues that arise in environments with no static types. In the same vein, we also seek to help scientists reason about atypical forms of types, such as units, that may occur throughout their computations.

To achieve these goals, we designed several new language features. These include a number of new patterns, such as the `annotate` pattern, which allows pattern matching on specially formatted MATLAB comments. We also introduce `type` and `dimension` patterns, which match join points corresponding to a particular MATLAB base type or array size respectively. In addition, we now allow for context exposure of the loop body in all patterns which match loops. This enables ASPECTMATLAB programmers to implement aspects which rewrite and modify the loop body.

Another important challenge was to improve the performance of ASPECTMATLAB code. By inserting aspect code directly into the base MATLAB code, as well as making local copies of aspect properties we eliminate a significant amount of overhead.

1.1 Contributions

The major contributions of this thesis are as follows:

Extensions to ASPECTMATLAB **language:** Building upon the ASPECTMATLAB language, we introduce a number of language extensions which add more expressiveness. These include the introduction of new patterns, such as the `annotate` pattern, as well as new possibilities for action code.

Improvements to the ASPECTMATLAB **compiler:** Several functions of the ASPECTMAT-

LAB compiler have been reworked, in order to support not only new language features, but to improve general performance.

Scientific aspects: We have developed a library of new aspects which provide easily acquired benefits to ASPECTMATLAB users.

Experimental evaluation: We have tested the ASPECTMATLAB language on a set of benchmarks to demonstrate the feasibility of woven code and the importance of our optimizations. Aspects are shown to have a reasonable amount of overhead.

1.2 Thesis Outline

This thesis is divided into 8 chapters, including this introduction chapter. In *Chapter 2*, we provide an introduction to the original ASPECTMATLAB language, describing it's language structures and their implementation. *Chapter 3* discusses the various extensions we have made to the basic ASPECTMATLAB language, and the motivation behind these extensions. *Chapter 4* describes various difficulties involved with extending MATLAB for aspects, and how we approach these difficulties differently from the original ASPECTMATLAB language. Here, we also detail various analyses which improve the performance of aspect code. *Chapter 5* showcases a number of use cases that demonstrate the importance of the proposed language extensions. The viability of ASPECTMATLAB is demonstrated in *Chapter 6*, where it is shown to perform effectively in comparison to a suite of benchmark MATLAB code. This chapter also demonstrates the significance of various implementation changes over the previous iteration of ASPECTMATLAB. *Chapter 7* discusses related work, and the ways in which our approaches differ. *Chapter 8* presents our conclusions, and provides an outline of possible future work.

Chapter 2
Background

2.1 MATLAB Types Overview

MATLAB, unlike more many more formal programming languages, does not have any static type declarations or static type checking. Instead, variable types are determined dynamically at runtime. Also dissimilar from most languages, the default type for storing information in MATLAB is a matrix, with even scalar constants being stored as a 1x1 array. Matrices have two important characteristics, their dimensions, representing the size of the matrix, and their base type, which represents the variety of data the matrix is allowed to hold. The data type of a MATLAB variable can be one of several default types, such as the numeric types double or int32, or they can be user-defined.

In MATLAB, type information is determined at run-time. Due to the lack of type declarations, as the program progresses, the same variable may contain values of different types. This lack of static type declarations is very convenient for fast prototyping, as it allows programmers to write code without having to consider the minute details of their program. However, despite the convenience this provides, MATLAB programmers often do have an idea as to what base types and dimensions their variables are expected to have throughout their program. Despite the fact that no formal type declarations exist, the authors of many MATLAB functions leave comments which specify the types expected of its arguments. An example of this variety of informal declaration is shown in *Figure 2.1*. While this information is not leveraged by MATLAB itself, failure to meet these informal recommendations can result not only in run-time errors, but also incorrect results. Type errors of this variety can easily propagate throughout a program, making it difficult to determine from where the error originated. In this regard, MATLAB's convenience comes with a trade-off, and when choosing MATLAB, programmers forgo the benefits that static types provide.

```
function [F, V] = nbody3d(n, R, m, dT, T)
%-------------------------------%
% This function M-file simulates the gravitational
%   movement of a set of objects
% Invocation:
%   >> [F, V] = nbody3d(n, R, m, dT, T)
%   where
%   i. n is the number of objects,
%   i. R is the n x 3 matrix of radius vectors,
%   i. m is the n x 1 vector of object masses,
%   i. dT is the time increment of evolution,
%   i. T is the total time for evolution,
%
%   o. F is the n x 3 matrix of net forces,
%   o. V is the n x 3 matrix of velocities.
%-------------------------------%
```

Figure 2.1 Header of MATLAB simulation of n-body problem [RP99]

2.2 ASPECTMATLAB

In ASPECTMATLAB, aspects were developed as an extension to object-oriented MATLAB code. Object-Oriented MATLAB classes are allowed to contain a `properties` block, where data that belongs to an instance of the class is defined. These properties can be defined with default values or initialized in the class constructor, and can consist of either a fixed set of constant values, or depend on other values, and be evaluated when required. Object-Oriented MATLAB classes also allow for a `methods` block, which can include class constructors, property accessors, or ordinary MATLAB functions. Methods and properties can be declared public, protected, or private.

ASPECTMATLAB expands upon this by adding aspects. Aspects are an extension to the base MATLAB grammar, and like a MATLAB class, an aspect is a named entity, which has a body. The body of an aspect not only allows for the properties and methods constructs, but also allows for two aspect-related blocks: `patterns` and `actions`. Patterns, which are analogous to pointcuts in other aspect-oriented languages, are used to pick out sets of join points in program flow. Actions, which are analogous to advice, are blocks of code that are intended to be joined to specific points of the base program. Actions specify what should be done when code is matched by patterns.

In *Figure 2.2* we see an example which makes use of these four features of aspects. The `properties` block defines a counter, which is initialized at its declaration and can be used throughout the aspect. The `methods` block defines a function called increment. In the `patterns` block, we define a pattern,called callAdd, that we want to match in the base code. In this case, we match calls to the function add. Finally, the `actions` block defines an action called actCall. This action specifies that we should call the method increment after every join point in the base code which matches the pattern callAdd. It then displays the name of the function.

Patterns, which must be contained in the `patterns` block of an aspect, are formed by a unique name and a pattern designator. The pattern designator can consist of one of ASPECTMATLAB's several primitive patterns, each of which target specific MATLAB constructs, or it can be a logical combination of them. ASPECTMATLAB was introduced with a variety of primitive patterns to match basic language constructs. An emphasis was made on patterns to match the cross-cutting concerns found in a scientific programming language. Since MATLAB code relies heavily on array constructs, and many programs are written as large functions containing several loops, array accesses and loops are key targets for patterns.

There are several function matching patterns, `call`, `execution`, and `op` which match calls to a specified function, the execution of a specified function, and calls to basic operations respectively. These patterns all take as a parameter the name of the function or operation they should match. For example, in *Figure 2.2*, we see that the `call` pattern takes as a parameter 'add', meaning that it will match calls to the function add.

ASPECTMATLAB has two array-related patterns `get` and `set`, which allow for match-

```
aspect myAspect
properties
   counter = 0;
end

patterns
   callAdd : call(add);
end

methods
   function increment(this)
       this.counter = this.counter + 1;
   end
end

actions
   actCall : after callAdd : (name)
       this.increment();
       disp(['calling ', name]);
   end
end

end
```

Figure 2.2 Simple ASPECTMATLAB example

ing accesses of and assignments to arrays respectively. Similarly to the function matching patterns, the `get` and `set` patterns take as a parameter the identifier of the variable they should match. Since an array access can exist within another array access, it is important to note that patterns are matched in order of evaluation of an expression, meaning patterns matching sub-expressions are evaluated before the containing expression.

ASPECTMATLAB also features loop-related patterns `loop`, `loophead`, `loopbody` which allow for matching on various portions of loops in MATLAB. Unlike functions and array accesses, there is no easy way to identify specific loop join points in the base code. Instead, loops are specified by the name of the loop iterator variable.

Finally, the `within` pattern allows for limiting the context with which matches can be made. It takes as a parameter the type of construct to match within, such as functions or loops, as well as an identifier, which can be used to specify a particular function, loop or other construct. This pattern matches all join points within the construct, and thus can be used in conjunction with other patterns to restrict the scope of matching.

In order to match more complex patterns, ASPECTMATLAB allows for compound patterns to be created using logical combinations of primitive patterns. An example of this is shown in *Figure 2.3*. `pCallTest` matches all calls to the function test that occur within a loop, `pGetOrSet` will match array access and assignments occuring within a function `add`, and `pCallExec` matches the previously defined pattern `pCallTest` as well as the execution of the test function itself.

```
patterns
  pCallTest : call(test) & within(loops, *);
  pGetOrSet : (get(*) | set(*)) & within(function, add);
  pCallExec : pCallTest | execution(test);
end
```

Figure 2.3 Compound ASPECTMATLAB patterns

There are three types of actions in ASPECTMATLAB, `before`, `around`, and `after`, which specify when, in relation to a matched join point, a piece of code should be executed. As one might expect, `before` actions are woven directly before a join point, and `after` actions are woven directly after a join point. The third type of action, `around` actions, are different in that they replace the join point completely. In order to execute the join point itself when using an `around` action, a special `proceed` call exists. This call can be used in the action code to execute the original join point. Omitting this call from action code results in the original join point never being executed.

ASPECTMATLAB allows for extraction of context-specific information about join points via the use of pre-defined context selectors. These selectors are specified along with an action definition, and allow for context-specific information to be used within action code. An example of context exposure is shown in *Figure 2.2* on line 17, where we use the name selector, to expose the name of the function matched by the pattern. This information is then used on line 19 to display the name of the function being called. The applicable selectors depend upon the type of join point being matched.

Chapter 3
AspectMatlab++ Language Extensions

We have defined and implemented a variety of new patterns to match additional language constructs. Given that our target audience includes novice programmers, we wanted to include a pattern in AspectMatlab that allows Matlab programmers who may not be familiar with aspect oriented programming to exert more control as to where aspect code would be woven. To this end, we introduce the `annotate` pattern, which allows for pattern matching on special Matlab comments. This pattern makes a useful tool for our own aspects, as it allows for general solutions to problems to be written as aspects, while making it possible for the specifics to be written into the base code as part of annotations.

To address the difficulties that Matlab's dynamic typing we introduced patterns for matching based on the size of arrays with the `dimension` pattern, as well as the type of data that arrays store, with the `type` pattern.

Another important extension was to allow for context exposure of the loop body in loop patterns through use of a special `body` call. This is essential for `around` advice on loop pattern, which themselves are of the utmost importance when targeting scientific languages, as it makes it possible for us to alter the loop body while still executing the base code correctly.

This chapter details the specification and basic use of these extensions. Advanced uses are demonstrated in *Chapter 5*, where they are used to create general-use scientific aspects.

3.1 Annotation Pattern

The annotation pattern differs from other patterns in AspectMatlab in that it does not match on Matlab code itself. Instead, we allow for programmers to write annotations which take the form of structured comments in their base code. The `annotate` pattern then matches these specially formatted comments. This makes it possible to provide information to the aspect program at any point in the execution, and allows for code to be woven easily into arbitrary points of a program, all without requiring any alteration to the execution of the base code. This new functionality makes it easy to write code with aspects in mind, by allowing for easy communication of relevant information from the base program to the aspect.

Due to the fact that these annotations are not be executed by a Matlab runtime, this approach has the benefit of ensuring that the program executes normally without having to weave aspects. The syntax for an annotation is shown in *Figure 3.1*. To specify that a particular comment should be recognized as an annotation, it is marked using the '`@`' symbol, and is followed by an identifier that gives the name of the annotation. Following the identifier is a list of arguments, whose values can be exposed as context in an action definition.

```
<annotation> ::=
    '%@' <annotation_name> <annotation_arguments>*
  | '%@' <annotation_name>
      <annotation_arguments>* '%' <comments>*

<annotation_arguments> ::=
    IDENTIFIER | STRING_LITERAL | CONSTANT
  | '[' <array_argument> ']'

<array_argument>::=
    <annotation_argument>
  | <array_argument> ',' <annotation_argument>
```

Figure 3.1 Syntax of Matlab Annotation

There are four types of arguments that can be exposed as context, `var` (IDENTIFIER), `str` (STRING_LITERAL), `num` (CONSTANT), and arrays of other arguments. Exposure of a `var` provides the value of that variable as context to the aspect code. If a variable is undefined, it will instead be exposed as a value of class `AMundef`, an empty MATLAB class. `str` exposes a string, and `num` a numeric value as a double. Arrays of arguments expose a cell array containing the context exposed by those arguments.[1] All arguments adhere to standard Matlab syntax.

The syntax for the annotation pattern is shown in *Figure 3.2*. It takes the name of the annotation it should match, as well as a list of arguments that are expected to be present in the annotation. In the event that a pattern matches the identifier and arguments, but has arguments in excess of those specified, the pattern will still match, and the excess arguments will not be exposed as context. This allows a matlab programmer to easily integrate annotations into their comments, without having to omit information to fit the annotation formatting. The allowed arguments are those described above and an insufficient number of arguments will result in no match.

```
<annotation_pattern> ::=
      'annotate' '(' <annotation_name>
             '(' <annotation_pattern_selector>* ')'')'
<annotation_pattern_selector> ::=
      'var' | 'str' | 'num'
    | '[' <annotation_pattern_selector>  ']'
    | '*'
```

Figure 3.2 Syntax of Annotation Pattern

An example of use of the annotation pattern is shown in *Figure 3.3*. In this case, the pattern matches all annotation comments that begin with the name ‘type’ and have at least three arguments: the first a variable name, the second a string, and the third an array of numbers.

[1]Cell arrays in MATLAB allow for different elements in the array to have different types, and so can be used to represent heterogenous collections.

```
patterns
  typepat : annotate(type(var, char, [num])) ;
end
```

Figure 3.3 Example of Annotation Pattern

3.2 Type Pattern

The `type` pattern, introduces matching on the base type of arrays to AspectMatlab. For this pattern, a base type can be one of several Matlab defaults, such as double, string, int32, or it can be a user defined class type. The `type` pattern captures all variable accesses and assignments in which the variable matches a specified Matlab base type. The syntax for invoking the type pattern is `type(<basetype>)`, where `basetype` is the name of the Matlab base type to be matched. Accesses and assignments are captured in the order of evaluation of an expression, with sub-expressions being matched before their containing expression. For assignments, the Matlab type considered is the one which would be held after assignment occurs.

To match accesses or assignments of specific arrays, but only when they are of a particular type, one can use a compound pattern of the `type` and `get` or `set` patterns. Examples of `type` patterns are given in *Figure 3.4*. Pattern `isint32pat` matches all join points where an array access or assignment is being performed involving an array of type int32. Pattern `isxsinglepat` demonstrates a compound pattern using `type`, `get`, and `set` to match all accesses and assignments to array `x`, when `x` is of type single.

```
patterns
  isint32pat : type(int32) ;
  isxsinglepat :
    (get(x)|set(x))&type(single) ;
end
```

Figure 3.4 Example of Type Pattern

In addition to the standard MATLAB base types, we have defined one further base type called `realint` to capture the special issue in MATLAB with its use of positive real integers. As the default data type in MATLAB is a double, all arrays can be indexed using double values. However, in the event that this double value does not correspond to a positive integer, this returns an error. To determine if double `x` is a real integer or not, it is necessary to check that `x = round(x)`. The inclusion of this base type makes it easy to determine whether the value in a matched array can be used as an index.

3.3 Dimension Pattern

Due to the fact that arrays are the default data type in MATLAB, it can be helpful to identify arrays by their size and shape. To this end we introduce the `dimension` pattern. The `dimension` pattern is similar to the `type` pattern, but instead of matching join points based on type, it instead matches by the dimensions of the associated vector.

The `dimension` pattern takes as arguments the size of the dimensions of the matrix it should match. Similarly to the `type` pattern, the `dimension` pattern matches variable accesses and assignments when the array is of the shape specified by the pattern's arguments. The syntax for invoking the dimension pattern is `dimension(<dimensions>)`, where `dimensions` is a comma separated list of the expected dimensions of the array. For each dimension, an integer value corresponding to the expected size of the dimension may be specified, or the wild-card symbol, '`*`' may be used to indicate that a dimension can be any size. As with the `type` pattern, accesses of specific arrays can be accomplished by using a compound pattern of the `dimension`, `get` and `set` patterns.

Examples of the `dimension` pattern are shown in *Figure 3.5*. Pattern `dimp` will match all array accesses and assignments involving arrays that have 3 dimensions, and whose first dimensions is of size 2. Pattern `dimx2by2` will match array accesses or assignments to `x` when `x` is a 2 by 2 matrix.

```
patterns
  dimp : dimension(2,*,*) ;
  dimx2by2 :
    (get(x)|set(x))&dimension(2,2);
end
```

Figure 3.5 Example of Dimension Pattern

3.4 Loop Body Context Exposure

Loops are a critical structure to target when writing an aspect language for scientific programming. To ensure these constructs can be handled meaningfully, the original AspectMatlab introduced several patterns which matched on loops. While it did successfully introduce means of handling loops themselves, one significant shortcoming is that it did not introduce any means of handling or restructuring the bodies of loops. To allow AspectMatlab programmers to deal with the body of loops more precisely, we introduce the `body` call.

AspectMatlab features a special `proceed` call which can be used in `around` advice to execute the original join point. While useful in most scenarios, with loop patterns it would be useful to be able to execute simply the body of the loop, as opposed to the entire join point. The `body` call is similar to the `proceed` call, however, instead of executing the entire join point, it simply executes a portion of it - that portion which corresponds to the body of a loop. To interact with the contents of the body, we also introduce the `loopiterator` keyword, which can be assigned a value to replace or modify the value held by the loop iterator variable.

Example *Figure 3.6* showcases the use of these keywords, with an aspect that replaces loops which iterate over `i` with a loop which iterates over the square root of the base Matlab loop's arguments. The `args` exposed as context are the loop iteration space, which is then square rooted and passed in as a loop iterator to another for loop, which makes a call to `body` to execute the body of the original loop.

```
actions
  sqrtiter : around loop(i) : (args)
    for loopiterator = sqrt(args)
      body()
    end
end
```

Figure 3.6 Example of Loop Body Context Exposure

Chapter 4
AspectMatlab++ Engine Enhancements

While the original implementation of AspectMatlab provided a clean means for weaving aspect-oriented code into Matlab, the implementation, making use of Matlab classes, was particularly slow. Aspect code was converted into Matlab classes, with action code being converted into Matlab class methods. Properties, as well as methods used by aspect code remain part of the Matlab class after compilation. These properties and methods are then referenced from the appropriate join points in the base code. Unfortunately, object-oriented Matlab classes require a significant amount of overhead to access their methods and properties, enough to cause woven code to run many times slower than the base code on its own. In order to eliminate this slowdown of aspect code, we inline the action code, and make local copies of aspect properties to avoid unnecessary overhead.

To demonstrate our enhancements, we look at the weaving of a simple aspect which counts flops, shown in *Figure 4.1*, and apply it to a heat equation solver in *Figure 4.2*, the result of which is shown in *Figure 4.3* and *Figure 4.4*.

4.1 Inlining of Action Code

After weaving, the methods corresponding to action code would be called from the join points which match the associated patterns. This is a functional solution, however Matlab classes have a significant amount of overhead involved in their use. Calling a method from a Matlab class entails a significant amount more overhead than a typical Matlab function

```
aspect flopcount
   properties
      count = 0;
   end

   patterns
      flopp : op(*);
   end

   actions
      aflop: before flopp
         count = count + 1;
      end
   end
end
```

Figure 4.1 Simple aspect to count all floating point operations

```
function solveHeatEquation(a,steps)
  tN= 3; % end of time interval
  N = 300; % set total steps
  h = 2*pi/(N-1); % set spacial step
  X = [h:h:2*pi]; % set X axis points - the first point is ommitted (0)
  U0 = 0*X; % set initial condition
  U0(round(end/2.2):round(end/1.8)) = 1;
  D = (Dxx(N)/h^2); % set second spatial derivative matrix
  function y = F(t,u) % set rhs of ODE, i.e. Ut
    y = a*D*u;
  end
  W = RungeKutta4(@F,[0, tN],U0,steps,1); % find the solution
  disp('computation finished');
end
```

Figure 4.2 Matlab function which solves the heat equation

```
classdef flopcount < handle
  properties
    count = 0;
  end
  methods
    function [] = flopcount_aflop(this)
      count = (count + 1);
    end
  end
end
```

Figure 4.3 Generated code for the flopcount aspect for *Figure 4.1*

call. Looking at the woven code, shown in figure 4.4, it can be noted that there are many calls to the method `flopcount.flopcount_aflop()`.

To determine the impact of these calls to MATLAB classes, we test the overhead of the call itself by making calls to empty functions. In *Figure 4.1*, we show the time required for a call to an ordinary MATLAB function `empty()`. This is compared with calls to a class method, `empty(obj)`, which can also be invoked by calling `obj.empty()`. We also compare to calls to a static class method `test.empty()`. These results demonstrate that calls to object-oriented MATLAB code are more than 12 times slower than ordinary function calls, and that static methods are slower still.

	Time (s) for 1000000 calls	Time (μs) per call
empty()	0.104	1.04
empty(obj)	1.294	12.94
obj.empty()	1.922	19.22
test.empty()	2.473	24.73

Table 4.1 Function call overhead for methods in object-oriented MATLAB

To increase the performance of woven aspect code, it would be ideal to do away with these method calls. One possible solution would be to performing a simple inlining on the function calls woven into the base code. This would require further work to be done in order to inline around actions, where we have to deal with proceed calls that require

```
function [] = solveHeatEquation(a, steps)
  global AM_GLOBAL;
  if isempty(AM_GLOBAL)
    AM_GLOBAL.flopcount = flopcount;
    AM_EntryPoint_0 = 1;
  else
    AM_EntryPoint_0 = 0;
  end
  tN = 3;  % end of time interval
  N = 300;  % set total steps
  AM_tmpBE_0 = 2
  AM_GLOBAL.flopcount.flopcount_aflop();
  AM_tmpBE_1 = (AM_tmpBE_0 * pi)
  AM_GLOBAL.flopcount.flopcount_aflop();
  h = (AM_tmpBE_1 / (N - 1));  % set spacial step
  AM_tmpBE_2 = 2
  AM_GLOBAL.flopcount.flopcount_aflop();
  AM_tmpBE_3 = (AM_tmpBE_2 * pi)
  X = ([(h : h : (AM_tmpBE_2 * pi))]);  % set X axis points - the first
   point is ommitted (0)
  AM_tmpBE_4 = 0
  AM_GLOBAL.flopcount.flopcount_aflop();
  AM_tmpBE_5 = (AM_tmpBE_4 * X)
  U0 = AM_tmpBE_5;  % set initial condition
  AM_GLOBAL.flopcount.flopcount_aflop();
  U0((round((end / 2.2)) : round((end / 1.8)))) = 1;
  AM_GLOBAL.flopcount.flopcount_aflop();
  D = (Dxx(N) / (h ^ 2));  % set second spatial derivative matrix
  W = RungeKutta4(@F, [0, tN], U0, steps, 1);  % find the solution
  disp('computation finished');
  if AM_EntryPoint_0
    AM_GLOBAL = [];
  end
end
```

Figure 4.4 Example of code woven before inlining of action code

passing of context information. A simpler option is to weave the action code directly into the base code without ever writing it into the aspect class as a method.

The first step towards accomplishing this task is to perform a renaming on the action code, to ensure that operations local to the aspect do not overwrite variables in the base code. Once this is completed, the action code can be inserted directly into the AST. Assignments to set up context selectors used by the action code are inserted before the action code itself in the AST. The result of the inlining on our example is shown in figure 4.5, with all function call overhead having been eliminated.

4.1.1 Considerations for Around Advice

Special consideration must be taken for the case of around advice. Due to the fact that the original join point is replaced when using around advice, the original join point must either be removed from the AST, or placed at the location of a proceed call if one is made. In addition, we also have to consider the fact that around actions can return data, and the data returned should correspond to the result of the execution of the original join point. To accomplish these tasks, we replace proceed calls with the original join point, and set the result aside, to be returned as a result at the end of the around advice.

The `body` calls are handled similarly to `proceed` calls, with the difference being that we replace the call to `body` with the entirety of the body of the loop.

It is worth noting that the previous implementation's handling of around advice required passing of context information to action code. In order to ensure that the join point is correctly executed when a proceed call is made, it was necessary to store all necessary context information within the action method, and use a switch statement to refer to the context of specific join points. By writing the action code directly into the AST of the base code, instead of making a call to a class method, this overhead is unnecessary. We can simply execute the join point with its original context intact, resulting in further performance enhancements when several context selectors are used.

```
function [] = solveHeatEquation(a, steps)
  global AM_GLOBAL;
  if isempty(AM_GLOBAL)
    AM_GLOBAL.flopcount = flopcount;
    AM_EntryPoint_0 = 1;
  else
    AM_EntryPoint_0 = 0;
  end
  tN = 3;  % end of time interval
  N = 300;  % set total steps
  AM_tmpBE_0 = 2
  AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
  AM_tmpBE_1 = (AM_tmpBE_0 * pi)
   AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
  h = (AM_tmpBE_1 / (N - 1));  % set spacial step
  AM_tmpBE_2 = 2
  AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
  AM_tmpBE_3 = (AM_tmpBE_2 * pi)
  X = ([(h : h : (AM_tmpBE_2 * pi))]);  % set X axis points - the first
   point is ommitted (0)
  AM_tmpBE_4 = 0
  AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
  AM_tmpBE_5 = (AM_tmpBE_4 * X)
  U0 = AM_tmpBE_5;  % set initial condition
  AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
  U0((round((end / 2.2)) : round((end / 1.8)))) = 1;
  AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
  D = (Dxx(N) / (h ^ 2));  % set second spatial derivative matrix
  W = RungeKutta4(@F, [0, tN], U0, steps, 1);  % find the solution
  disp('computation finished');
  if AM_EntryPoint_0
    AM_GLOBAL = [];
  end
end
```

Figure 4.5 Example of code woven after inlining of action code

4.2 Local Copies of Aspect Properties

With action code having been inlined, aspect properties referenced in action code must still be accessed through the generated MATLAB class. While the inlining process does reduce a significant amount of the overhead involved in woven code, the overhead from accessing these object properties may still be significant. We carried out a study to determine the impact of property accesses and assignments by performing each 1000000 times within a loop. The results are shown in *Figure 4.2*. While not as significant as the overhead for calls to object-oriented methods, property accesses and assignments still take up more time than function calls to ordinary MATLAB code, and are much slower than assignments to local variables.

	Time (s) for 1000000 calls	Time (μs) per call
empty()	0.104	1.04
property access	0.366	3.66
property assign	0.286	2.86

Table 4.2 Property access and assignment overhead for object-oriented MATLAB

In order to further reduce the overhead due to frequent accessing of object properties it is helpful to make local copies of properties whenever possible. By doing this, base code has many matched join points, or action code which makes reference to the same property several times, can be rendered more efficient. One precaution we must take when making local copies is ensuring that the properties are updated before they are used or modified elsewhere in the code. To ensure this, we must copy back to the object prior to any function calls to code that is woven by the same aspect. Similarly, we must be certain to copy the property back to the associated object before the end of the woven function as well.

In order to accomplish this, we perform a local copy analysis, which is composed of two sub-analyses, which determines which local copies must be live after each program block. This analysis allows for us to copy back to the aspect prior to making a dangerous function call, and load a local copy only when necessary at the end of each program block. To set up the analysis, we mark all other function calls which are woven by the same aspect, such that we can recognize them as we traverse to code. The function calls are labeled as dangerous calls, for which we need to write our local copy back to the aspect. It is worth noting that by default the ASPECTMATLAB compiler assumes that aspect-woven code will not be called from non-aspect woven code. This allows for copies to persist through most function calls, increasing performance.

We then perform two analyses, the Local Copy Analysis to determine where local copies should be made, and the Local Write Analysis, to determine where copies should be written back. These analyses were written using the Matlab Static Analysis Framework, MCSAF [Doh11].

4.2.1 Local Copy Analysis

The Local Copy Analysis computes for each program point which property uses could made into a local copy. It is implemented using a forward flow sensitive analysis. At merge points we take the intersection of the merged sets. The main cases we consider for this analysis are as follows:

-Aspect Property Use: For each property use, the property being used and the line it is used on is added to the current set.

-Unsafe Function Call: Upon calling any unsafe function, as we have defined above, all items in the current set are killed.

-Clear workspace ('clear'): All items in the current set are killed.

The results of this analysis are used to determine optimal locations for local copies to be made.

4.2.2 Local Write Analysis

The Local Write Analysis computes for each program point which property writes could be written to at that point. It is implemented using a backward flow sensitive analysis. At merge points we take the intersection of the merged sets. The main cases we consider for this analysis are as follows:

-Aspect Property Assignment: For each assignment to an aspect property, the property being used and the line it is used on is added to the current set.

-Unsafe Function Call: As with the Local Copy analysis, upon calling any unsafe function, all items in the current set are killed.

-Clear workspace ('clear'): All items in the current set are killed.

The results of this analysis are used to determine optimal locations for writing local copies back to aspect properties.

Once the analyses have been completed, we insert local copies of aspect properties and writes of these copies back to the aspect properties at relevant locations, starting with those program points which would cover the most copies and writes. Uses of aspect properties are then replaced with these local copies.

The result of the local copies can be seen in figure 4.6. Before its first access, the property is stored in a local variable `AM_tmpcount`. All operations that would be performed on the aspect property are then performed on this local copy instead. Before the end of the function, the final state of this copy is stored back in the object property. In this example, which originally required 6 property accesses and 6 property assignments, we now only require 1 of each, significantly reducing unnecessary overhead.

```
function [] = solveHeatEquation(a, steps)
  global AM_GLOBAL;
  if isempty(AM_GLOBAL)
    AM_GLOBAL.flopcount = flopcount;
    AM_EntryPoint_0 = 1;
  else
    AM_EntryPoint_0 = 0;
  end
  tN = 3;  % end of time interval
  N = 300;  % set total steps
  AM_tmpBE_0 = 2
  AM_tmpcount = AM_GLOBAL.flopcount.count;
  AM_tmpcount = AM_tmpcount + 1;
  AM_tmpBE_1 = (AM_tmpBE_0 * pi)
  AM_tmpcount = AM_tmpcount + 1;
  h = (AM_tmpBE_1 / (N - 1));  % set spacial step
  AM_tmpBE_2 = 2
  AM_tmpcount = AM_tmpcount + 1;
  AM_tmpBE_3 = (AM_tmpBE_2 * pi)
  X = ([(h : h : (AM_tmpBE_2 * pi))]);  % set X axis points - the first
   point is ommitted (0)
  AM_tmpBE_4 = 0
  AM_tmpcount = AM_tmpcount + 1;
  AM_tmpBE_5 = (AM_tmpBE_4 * X)
  U0 = AM_tmpBE_5;  % set initial condition
  AM_tmpcount = AM_tmpcount + 1;
  U0((round((end / 2.2)) : round((end / 1.8)))) = 1;
  AM_tmpcount = AM_tmpcount + 1;
  AM_GLOBAL.flopcount.count = AM_tmpcount;
  D = (Dxx(N) / (h ^ 2));  % set second spatial derivative matrix
  W = RungeKutta4(@F, [0, tN], U0, steps, 1);  % find the solution
  disp('computation finished');
  if AM_EntryPoint_0
    AM_GLOBAL = [];
  end
end
```

Figure 4.6 Example of code woven after local copies of aspect properties have been made

Chapter 5
Aspect Examples

Using the language extensions outlined in the previous chapter, we designed and implemented a number of aspects to help programmers to better understand and work with their MATLAB code. In order to ensure programs meet expected type requirements, we introduce a *type checking aspect*, which allows for programmers to specify stricter types using type annotations which are matched by the `annotate` pattern. The *unit checking aspect* uses unit annotations to allow programmers to ensure that their units match, following basic rules of dimensional analysis. This allows for type checking of a scientific variety. In order to better deal with peculiarities of MATLAB, we also introduce two *type profiling aspects*, which employ the `type` and `dimension` patterns to detect dynamic type information. To help programmers get the most out of MATLAB code, we also use aspects to automate loop transformations which use the newly introduced body context exposure for loops, improving the ability of ASPECTMATLAB to help programmers increase the efficiency programs by making *loop unrolling* and *loop reversal* optimizations quickly and easily. The full aspect code of each scientific aspect can be found at `http://www.sable.mcgill.ca/mclab/aspectmatlab/`.

5.1 Stricter type checking

In MATLAB, variable types are not declared, and are instead determined dynamically. This is beneficial for fast prototyping, as it allows programmers to focus on the task at hand.

Despite MATLAB's lack of static type checking, programmers often have some assumptions about the types being used in their programs, as shown previously in *Figure 2.1*. Using mechanisms provided by MATLAB in order to ensure that the types of variables are correct after every assignment would require the programmer to insert checks manually, a tedious job which opens up room for errors. The significant number of superfluous checks would negatively impact the performance.

Our solution to this problem is to leverage the power of the newly introduced `annotate` pattern to allow programmers to format their comments in such a way that the type information they contain can be confirmed by an aspect. *Figure 5.1* shows an example of what annotated code looks like. It is very similar to the original comments, but by formatting the comments into type annotations, the information in the comments can be used by the ASPECTMATLAB compiler.

```
function [F, V] = nbody3d(n, R, m, dT, T)
%--------------------------
% This function M-file simulates the gravitational
%   movement of a set of objects
% Invocation:
%   >> [F, V] = nbody3d(n, R, m, dT, T)
%   where
%inputs:
%@type n 'double' [1,1] %number of objects
%@type R 'double' [n,3] %matrix of radius vectors
%@type m 'double' [n,1] %vector of object masses
%@type dT 'double' [1,1] %time increment of evolution
%@type T 'double' [1,1] %total time for evolution
%outputs:
%@type F 'double' [n,3]  %matrix of net forces,
%@type V 'double' [n,3]  %matrix of velocities.
%-------------------------
```

Figure 5.1 Header of MATLAB simulation of n-body problem with type annotations

The formatting of these annotations are specified by the `annotate` pattern in our type checking aspect and operates as shown in *Figure 5.2*. As with all ASPECTMATLAB annotations, the first piece of information is annotation name, "type". After the annotation name, this particular annotation has three other arguments. These are the identifier for the variable it should be checking, a string which declares the base type the variable is expected to have, and an array that lists the expected size of each dimension.

```
⟨type annotation⟩ ::⇒
    '%@' 'type' ⟨variable⟩ '⟨type⟩'  '['⟨dimensions⟩']'  ⟨comment⟩*
⟨type⟩ ::⇒
    ⟨base matlab datatype⟩
    | ⟨user defined datatype⟩
⟨dimensions⟩ ::⇒
    ⟨dimension⟩  ',' ⟨dimensions⟩
    | ⟨dimension⟩
⟨dimension⟩ ::⇒
    IDENTIFIER | DOUBLE | CHAR
```

Figure 5.2 Definition of a type annotation

As an example, the annotation `%@type n 'double' [1,1]` is checking that the variable n is of type double, and is a scalar (or 1 by 1 matrix). For the base type, the user may require that the variable be of any base MATLAB datatype, as well as any user defined datatypes. The dimensions can be expressed in one of three ways, which capture the variety of possible requirements a programmer may have of their program. The simplest method of communicating the dimensions is by simply specifying the expected size numerically in the annotation. In this case, the aspect will ensure that the size of the variable's dimension matches the one specified by the programmer. This is demonstrated on line 1 of *Figure 5.3*. The second method for specifying dimension size is by using an identifier which corresponds to a variable in the base MATLAB code, which contains the size it should match. This is shown on line 2 of *Figure 5.3*, where variable `b` is required to be a matrix of size n by m, where n and m will have been previously initialized in the base MATLAB code. Alternatively, a string can be used to specify the dimension sizes, as shown on lines 3,4 and 5 of *Figure 5.3*. When a string is used, the aspect checks that the size of the dimension matches any other dimensions which use the same string. For example, variable `c` is required to have 2 dimensions, and because both are specified by the same string, both dimensions must have the same size. Similarly, the annotations on lines 3 and 4 specify that the first dimension of `d` must be the same size as the second dimension of `e`, and vice versa.

```
%@type a 'double' [2,3,4]
%@type b 'userdef' [n,m]
%@type c 'char' ['x','x']
%@type d 'double' ['y','z']
%@type e 'double' ['z','y']
```

Figure 5.3 Example of type annotations

Once the aspect has been woven, the aspect code will check whether or not the types hold at the program point where the annotation is present. In addition, any time a variable is assigned to after the annotation, the aspect code will check again to ensure that it obeys the specified restrictions. These annotations are simple and straightforward and are able to be easily inserted anywhere in a program where a programmer is uncertain. They also correspond closely to comments that are typically found in MATLAB programs.

The type checking aspect, outlined in *Figure 5.4*, takes advantage of the annotation pattern and weaves type checking code around these annotations. There are only two patterns required for this aspect. The pattern `typeAnn` matches the type annotations. It matches annotations with the annotation name type, and takes three arguments, an identifier corresponding to the variable in question, a string constant corresponding to its type, and an array which can contain any combination of numerical constants, identifiers, or string constants, corresponding to its dimensions as described above. This demonstrates that while the annotation pattern is simple to use, it can be very powerful, allowing for easy introduction of language features that don't otherwise exist in MATLAB.

From here, the code which carries out type checking is straightforward. The action `actAnn`, given in *Figure 5.5*, uses the information provided by the annotation. This information is extracted using the `args` context selector, which yields a cell array of the arguments' values, as well as the `rep` context selector, which provides a cell array of string representations of the arguments' values. Before we type check, we confirm that the variable in question has been assigned a value. If it does not, the content of the variable will be of type `AMundef`, signifying that no definition has been provided and that we can proceed to storing type information. If the variable in question has been defined, we check that its

```
aspect typechecking
  ...

  patterns
    typeAnn : annotate(type(var, char, [*]));
    arraySet : set(*);
  end

  actions
    actAnn : before typeAnn : (args, rep, line)
      value = args{1};
      varname = rep{1};
      expectedtype = args{2};
      expecteddims = args{3};
      %Check that the provided variable meets the
      %specified type and dimensions
      %Store the type and dimensions for future checks
       ...

    actArraySet : after arraySet : (newval, name, line)
      %Compare with known type and dimensions for the
      %given variable name
      ...
  end
end
```

Figure 5.4 Outline of type checking aspect

value has the expected number of dimensions, the correct type, and the correct size for each dimension as specified by the annotation. In the event that there is a conflict, an error is emitted. If a string has been used to specify the size of a dimension, we check to see if any previous annotations used the same string - comparing size with the previous use if it has, and associating the current value for our variable if it hasn't. Regardless of whether type checking is performed, the information from the annotation is stored in `container.Map` objects, so future array assignments can be checked. The action `actArraySet` , which is woven around assignments to arrays, checks whether or not the variable being assigned to has an existing mapping from a previous type annotation. If it does, type checking is performed using similar checks to those made in the `actAnn` pattern, and throwing an error if the checks fail.

```
actAnn : before typeAnn : (args, rep, line);
  %context expose of args gives the arguments provided by the
  %annotation in order
  %context expose of rep gives the string representation of the
  %annotation arguments, which we use to obtain a string
  %representation of the variable
  value = args{1};
  varname = rep{1};
  expectedtype = args{2};
  expecteddims = args{3};

  %if the variable is defined at the point of the annotation, we
  %immediately perform type checking
  if(¬isa(value,'AMundef'))
    dimensions = dims(value);
    if((ndims(value) ≠ size(expecteddims,1)) )
       %prepare error message
    end
    if(¬isa(value,expectedclass))
      %prepare error message
    end

    for dim = 1:size(expecteddims,1)
      if(isa(expecteddims{dim},'char')
        %check against any previous definition for the string
      if(ismember(expecteddims{dim}, keys(chardims)))
        if(size(value,dim)≠ chardims(expecteddims{dim}))
          %prepare error message
        else
          %associate value with string if no previous definition
          %exists
          chardims(expecteddims{dim}) = size(value,dim);
        end
      elseif(size(value,dim) ≠ dimensions{dim})
        %prepare error message
      end
    end
    %emit error message if one exists
  end
  %Whether the variable has been defined, we associate the
  %variable name with the expected class and dimensions
  varclass(varname) = expectedclass;
  vardims(varname) = expecteddims;
end
```

Figure 5.5 actAnn action of typechecking aspect in detail

Using this aspect, it is possible to leverage the types specified by the programmer, and throw meaningful errors when they are not met. Due to the fact that types are stored as annotations, it is possible for a programmer to simply run their program without weaving type checking code once they are certain that their program will execute correctly. This allows for programmers to enable type checking at a small cost in performance to ensure their MATLAB code executes as expected, and later dispense with the type checking to ensure optimal performance. These annotations can be introduced into a program without knowledge of aspect oriented programming, nor significant knowledge of the inner workings of ASPECTMATLAB. As a result, this aspect is useful even for those who are not interested in learning the ASPECTMATLAB language.

5.2 Units of measurement as types

In the previous section we explored an aspect for handling traditional type checking, however, for scientific programmers there exists more type information presented in *Figure 2.1* aside from the dimensions of variables. Many inputs may have associated units of measurement, corresponding to physical qualities such as times, distances, and masses. Even in a programming language such as MATLAB, which targets scientific programmers, these units of measurement will be lost, as it is most easy and efficient to store data as doubles instead of having a separate class for every piece of data. This is unfortunate, as these units of measurement have meaning within the context of a program, particularly programs with a large number of arithmetic operations. For example, we know that it makes sense to add a distance to a distance, and the result of such a computation would be a distance itself. However, it is not meaningful to add a distance to a mass, as the result of the computation would be meaningless. Similarly, assigning a value that is known to be a mass to a variable which should contain a distance would also be incorrect. By keeping track of the units that variables are intended to have, it is possible to prevent programmers from making mistakes by throwing errors when incorrect units are used. In this sense, units can be thought of as types unto themselves. By providing programmers with a means to incorporate units into their programs, it can be ensured that their programs only use them safely.

In order to take advantage of unit information, we introduce the idea of unit annotations.

Figure 5.6 shows how unit annotations can be used to take advantage of the information programmers provide. Similarly to the type aspect, our unit aspect defines a formatting which unit annotations must follow, and this is shown in *Figure 5.7*. Following the annotation name, two arguments are required. The first is the string corresponding to the variable whose units we are keeping track of. The second is an array of strings containing the SI units of the variable. For example, the annotation `%@unit 'dT' ['s']` specifies that the variable dT is a measure of time, in seconds. We support several common derived units, and for those unsupported, combinations of the 7 base SI units can be used. For example, a newton which could be conveyed in an annotation as `['N']`, may also be expressed as $kg \cdot m/s^2$ which could be written in an annotation as `['kg','m','s^-2']`. Exponents can be used in unit annotations, and units may be provided in any order.

```
function [F, V] = nbody3d(n, R, m, dT, T)
%--------------------------
% This function M-file simulates the gravitational
%   movement of a set of objects
% Invocation:
%   >> [F, V] = nbody3d(n, R, m, dT, T)
%   where
%inputs:
%@unit 'n' [] %number of objects
%@unit 'R' ['m'] %n x 3 matrix of radius vectors
%@unit 'm' ['kg'] %n x 1 vector of object masses
%@unit 'dT' ['s'] %time increment of evolution
%@unit 'T' ['s'] %total time for evolution
%outputs:
%@unit 'F' ['N']
%@unit 'V' ['m','s^-1']
%-------------------------
```

Figure 5.6 Header of MATLAB simulation of n-body problem with unit annotations

The unit checking aspect, outlined in *Figure 5.8*, works similarly to the type checking aspect, taking advantage of these annotations the programmer can use in their code. The pattern `unitAnn` is a usage of the `annotate` pattern which matches all unit annotations, as they have been specified. The action `unitAnnAct` specifies that, around these annotations, we weave code which stores the type information and ensures that there is no conflict

```
⟨unit annotation⟩ ::⇒
    '%@' 'unit' '⟨variable⟩' '[' ⟨units⟩ ']' ⟨comment⟩*
⟨units⟩ ::⇒
    ⟨unit⟩ ',' ⟨units⟩
    | ⟨unit⟩
⟨unit⟩ ::⇒
    | '<si_unit>'
    | '<si_unit> '^' INTEGER '
⟨<si_unit>⟩ ::⇒
    m | kg | s | A | K | mol | cd | N | J | ...
```

Figure 5.7 Definition of a unit annotation

with the current units held in the variable and the ones the annotation specified. We apply these units as specified by the annotation when they are accessed by using the `get` pattern, and store these units in a struct which contains both the units as well as the original value. The original value is stored in a val field, and the units are stored in a unit field as an array containing the exponents of each of the 7 base SI units. For example, a value of 3 newtons would be stored as `struct('val',3,'units',[1,1,-2,0,0,0,0])`. Wrapping all units in such a structure would result in disruption of the normal program execution. To handle this, we perform a unit removal whenever we cannot be certain that such a struct would be handled appropriately, for example when it is used as a parameter in function calls, or used as an index for an array. We also match all arithmetic operations, and, using separate actions for each operation, determine what the units will be after a line of code is executed. In the case of addition, subtraction and exponentiation, we also ensure that there is no conflict in the units of the operands. From the point of a unit annotation onward, any addition or subtraction operations between a specified variable and another variable that has not been annotated with the same units will result in an error, and the programmer will be informed of the mismatch. Similarly, exponentiation performed with an exponent that has units will also result in an error. Using the `set` pattern, we check on each assignment whether a unit annotation has been violated, throwing an error when an assignment is made to that variable that has units other than those specified.

Because the units have been written in the program as annotations, the programmer is

```
aspect unitchecking
  ...

  patterns
    unitAnn : annotate(unit(char, [char]));
    pMlabConstructs : within(call,*) | within(get, *) |
   within(condition, *) | within(logical, *) | ...
  end

  actions
    unitAnnAct : before unitAnn : (args)
      %Store the variable name and units for future
      varunits[args{1}] = args{2}

    actAddOp : around op(+) : (args, line)
    actSubOp:  around op(-) : (args, line)
      %Compare the inputs of addition or subtraction, if
      %they conflict throw an error
    actMultOp : around op(*)  : (args)
    actDivOp :  around op(/)  : (args)
      %Determine what the units will be after
      %multiplication or division
    actExpOp : around op(^) : (args, line)
      %Throw error if exponent is not unitless,
      %determine what units will be after exponentiation

    addUnits : around get(*) & ¬pMlabConstructs : (name, obj)
      %if units would not interfere with normal expecution of
      %the program, we ensure that units are applied based on
      %name of variable, if an associated annotation exists.

    removeUnits : around get(*) & pMlabConstructs : (name,obj)
      %remove units if they would interfere with normal
      %execution of the program.

    actSet : after set(*) : (name, newVal, line)
      %Check that the units being assigned do not violate
      %annotation
  end
end
```

Figure 5.8 Outline of unit checking aspect

able to run the program without weaving in the aspect once they are satisfied that their program runs as expected. This means they can choose to run the program with the unit check on, ensure no unit violations take place, and then run the program without the aspect afterwards to avoid the overhead associated with unit checking.

5.3 Profiling real positive integers

The default numeric data type in all MATLAB programs is a double-precision floating point. This has some advantages for scientific programmers, as doubles are more likely to be used in most scientific computations. Not having to worry about conversions between reals and integers allows scientific programmers to focus on the functionality of their program. One negative side effect of this, however, is that all arrays in MATLAB can be indexed using real numbers. Given that indexing with a non-integer value is not meaningful, MATLAB only allows certain classes of real numbers to be used for indexing, “real positive integers".

A *real positive integer* is any real positive number which is equal to the result of rounding said number. Checking whether or not this property holds at all program points is a tedious task for a programmer. However, making a mistake in using a real that is not a real positive integer could result in unexpected runtime errors. In addition, it presents some difficulty for those interested in translating MATLAB programs to other languages, as it cannot necessarily be checked statically. To aid these two groups, an aspect was created to determine which variables cannot be safely used for indexing.

This aspect, outlined in *Figure 5.9* takes advantage of the type pattern, matching on all array accesses which do not contain real integers. By taking note of the variables which are not real integers, we can know which variables would be unsafe to use for array access operations. The pattern `realint` matches all array accesses which are not real positive integers. The action `actRealInt` uses the `name` and `line` context selectors to store the name of the variable being accessed, as well as the line number at which it was accessed. At the end of the execution of the program, the `actRealInt` action prints the result of the profiling, and informs the programmer as to which variables were unsafe, and the lines where they were found to be unsafe.

```
aspect safeindex
  properties
  nonrealint = container.Map;
  end

  patterns
    realint : get(*)&type(¬realinteger);
    exec : mainexecution();
  end

  actions
    actRealInt : after realint : (name,line)
      %Store the name and line of the variable, if it
      %has not already been stored.
      unrealintlines = nonrealint[name]
      if(¬ismember(unrealintlines,line))
        nonrealint[name] = [unrealintlines line]
      end

    results : after exec
     %Display results
      ...
  end
end
```

Figure 5.9 Outline of real integer profiling aspect

5.4 Profiling 1 dimensional arrays

Another area where profiling MATLAB is useful is array dimensions. As MATLAB is a matrix-based programming language, a one dimensional array can refer to either a matrix with only 1 column and several rows or a matrix with only 1 row and several columns. This differs from other more conventional programming languages, where information stored in a one dimensional array does not have an orientation. While the data contained in a one dimensional array may seem the same to a programmer regardless of orientation, the orientation can change the functionality of a program. An improperly oriented array could lead to program terminating runtime errors, or to data being improperly handled, causing incorrect results.

To aid programmers in understanding their code, we provide an aspect which profiles one dimensional arrays, as outlined in *Figure 5.10*. We match on all assignments of one dimensional arrays, and store information about the orientation and size of the array. At the end of execution of the program, the programmer is informed of the orientation of the arrays. Weaving and running this aspect provides the programmer with certainty that their arrays are handled correctly.

```
aspect profileoned
  properties
    columnvars = container.Map;
    rowvars = container.Map;
  end

  patterns
    onedcolumn : set(*)&dimension(1,*);
    onedrow : set(*)&dimension(*,1);
    exec : mainexecution();
  end

  actions
    actColumn : after onedcolumn : (name,line)
      %Store the name of the variable, the line it
      %occured on, and its orientation
      columnslines = columnvars[name]
      if(¬ismember(columnslines,line))
        columnvars[name] = [columnslines line]
      end

    actRow : after onedrow : (name,line)
      %Store the name of the variable, the line it
      %occured on, and its orientation
      rowslines = rowsvars[name]
      if(¬ismember(rowslines,line))
        rowvars[name] = [rowslines line]
      end

    results : after exec
      %Display results

  end
end
```

Figure 5.10 Outline of one dimensional array profiling aspect

The aspect itself consists of two primary patterns, both of which make use of the `dimension` pattern. The pattern `onedcolumn` catches all array assignments which correspond to a 1-dimensional column, and `onedrow` does the same for 1-dimensional rows. The actions `actColumn` and `actRow`, are performed after the assignments to the 1-dimensional array are made. The name and line context selectors are used so we can store information about the variable being assigned to, and the location in the code where the assignment took place. The action `results`, called after the execution of the program, prints out a list of assignments made to 1-dimensional arrays over the course of the program, consisting of the name, the line it occurred on, and the orientation.

5.5 Loop Transformations with Aspects

In addition to the aspects which help manage type information, we have also contributed new aspects with the goal of helping programmers improve the efficiency of their code. One such aspect is the loop unrolling aspect, outlined in *Figure 5.11*. Loop unrolling is a fairly simple transformation, in which the body of a loop is executed several times per loop iteration. The result is that the loop condition needs to be tested fewer times, and fewer jumps are required. The cost of this transformation is that the program's code size increases, and may experience slowdown due to poor register usage. However, unrolling loops by hand is tedious and time consuming, requiring large amounts of code to be copied and repeated. In addition, if loop unrolling decreases performance, additional effort must be expended to repair the code to its original state. This additional programming overhead decreases the appeal of this transformation.

In order to automate the process, and decrease the effort required to determine whether manual loop unrolling is a beneficial transformation, we introduce an aspect which unrolls for loops. This aspect takes advantage of newly introduced loop functionality, as well as annotations. In order to unroll the desired number of times, the unroll aspect takes information from an unroll annotation, which the programmer may place in their code, and which takes the form `@unroll 10`. It takes as an argument the number of time the loop should be unrolled, and from the point of the annotation onwards, for loops will be unrolled the specified number of times. Due to limitations of the AspectMatlab engine, the

```
aspect unroll
  properties
    numunroll = 2;
  end

  patterns
    annUnroll : annote(unroll(double));
    loopp : loopbody(*);
  end

  actions
    actUnroll : after annUnroll : (args)
    numunroll = args{1}
  end

  unroll2 : around loopp : (looptype, obj)
    if(strcmp(looptype,'for') && unrollvalue == 2)
      i = 0;
      while(i<(size(obj)-2)
        i = i+1;
        loopiterator = obj(i)
        body()
        i = i+1;
        loopiterator = obj(i)
        body()
      end
      while(i<size(obj)
        i = i+1;
        loopiterator = obj(i)
        body()
      end
    end
  end


  unroll5 : around loop : (looptype, obj)
   ...
  end

  ...

end
```

Figure 5.11 Outline of Loop Unrolling aspect

only accepted values for the unrolling parameter are 2,3,4,5, and 10. The `body()` call is used to duplicate the loop body multiple times. The result of weaving the aspect is that each loop will be unrolled to 5 extents, and at runtime, the loop which matches the desired number of unrolls will be run.

```
aspect reversal

  actions
    reversal : around loopp : (looptype, obj)
      if(strcmp(looptype,'for'))
        i = size(obj);
        while(i>0)
          loopiterator = obj(i)
          body()
          i = i-1;
        end
      end
  end
end
```

Figure 5.12 Outline of loop reversal aspect

Another loop transformation that can be performed with aspects is loop reversal. Like loop unrolling, loop reversal is a fairly straightforward, but performing it by hand on each loop to determine its value is time consuming. The transformation involves reversing the order in which values are assigned to the index variable. Again, we introduce an aspect which automates the process of reversing loops. This aspect functions by executing the body with a loop iterator that proceeds in reverse. This aspect, given in *Figure 5.12* and which involves only a few lines of aspect code, demonstrates the ease with which transformations can be executed.

Chapter 6
Performance

In this section, we provide the results of experiments performed on a set of 10 MATLAB benchmarks, which demonstrate the significance of the performance improvements made to the ASPECTMATLAB compiler, as well as the utility of the scientific aspects provided. The benchmarks used have come from various projects targeting MATLAB, such as FALCON [RP99] and OTTER [QMS98] , Chalmers University of Technology[1] and "The MathWorks' Central File Exchange"[2]. To analyze performance, we run each of the original benchmarks in MATLAB without weaving any aspects. We adjust the problem size of the benchmarks such that they take approximately 25 seconds to execute. We then weave code for each of the aspects outlined in chapter 4 using the ASPECTMATLAB compiler, and run each of them in MATLAB with the same problem size. Our performance analysis compares the running time of several benchmarks with each aspect. We then look at the impact of the optimizations outlined in Chapter 7 by performing the same experiment with them disabled. For our analysis, we use MATLAB version 2013a, and ran on an Intel Core i7-3820 CPU @ 3.60GHz x 8 processor and 16 GB memory machine running Ubuntu 12.04 LTS. To obtain our results, we test each individual benchmark 10 times and take the mean of the results.

[1] http://www.elmagn.chalmers.se/courses/CEM/

[2]http://www.mathworks.com/matlabcentral/fileexchange

6.1 Benchmarks

For our evaluation, we use a set of benchmarks retrieved from a wide variety of sources. They were selected both for their coverage of commonly used features in MATLAB as well as their applicability to the various scientific aspects they were to be tested on. In this section, we describe the benchmarks used.

beul

Solves the heat equation using the backward Euler method. Features numerous inputs to which units, types, and dimensions can be applied and verified.

crni

Uses the Crank-Nicolson finite difference method to find a solution to the heat equation. Suggests dimensions, and types and units can be applied.

diff

Calculates the diffraction pattern of monochromatic light through a transmission grating. Suggests units for inputs and variables throughout the program, and features nested loops.

fdtd

Applies the finite difference time-domain method to compute the electric and magnetic fields in a hexahedral cavity with conducting walls. Features several inputs and outputs to which units, types, and dimensions can be applied, and features several get and set operations in a single loop.

fiff

Calculates a solution to the wave equation using the finite difference method. Has numerous inputs to which types, and units can be applied, features nested loops and several array operations.

hnor

Normalises array of homogeneous coordinates to a scale of 1. Types and dimensions can be applied to the input and output argument, which are accessed throughout the program.

mils

Matlab package for solving mixed integer least squares problems. Features inputs and outputs for which types and dimensions are suggested.

nb1d

Computes the solution to an n-body problem in one dimension. Features nested loops, and some inputs for which types and dimensions can be applied, as well as variables which can be associated with units.

nb3d

Computes the solution to an n-body problem in three dimensions. Features nested loops, and inputs for which types and dimensions can be applied, as well as variables which can be associated with units.

capr

Computes the capacitance of a transmission line using finite difference and Gauss-Seidel iteration. A loop based program which features multiple inputs for which types and dimensions are suggested, and to which units can be applied.

6.2 Experimental Results

In Table 6.1, we list the execution time in seconds of different benchmarks. We then list the slowdown experienced when running benchmarks with each aspect. At the bottom of each column, we also provide the geometric mean of the slowdown across all benchmarks, giving the average slowdown one can expect to experience with each aspect. Aspects whose slowdowns are denoted by a "-" were not run with the particular benchmark. The reasoning for this is that it does not make sense to apply all aspects to all benchmarks - for example, the unit checking aspect cannot reasonably be used with a benchmark featuring no quantities with units.

	Time (sec)	**Slowdown**					
Benchmark	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling	Loop Unrolling	Loop Reversal
beul	24.95	1.65	3.78	1.42	1.29	-	-
crni	23.56	1.29	7.23	1.34	1.24	0.97	1.05
diff	23.82	3.20	3.75	1.39	1.28	1.18	1.11
fdtd	25.23	3.58	16.15	1.54	1.48	0.96	-
fiff	24.50	1.30	1.24	1.36	1.34	0.98	0.97
hnor	26.12	1.04	-	1.18	1.19	1.00	1.01
mils	25.05	1.17	-	1.55	1.52	-	-
nb1d	25.19	3.69	5.88	1.53	1.50	1.04	-
nb3d	24.67	3.53	6.60	1.53	1.42	1.08	-
capr	24.78	1.29	8.10	1.79	1.71	0.98	-
Geometric Mean		2.00	5.43	1.46	1.39	1.03	1.03

Table 6.1 Time in seconds for execution of benchmarks, and slowdowns with woven aspects

Given that most of the aspects are adding extra functionality to the base MATLAB program, we expect the runtimes to increase. From the results, we can note that both the type checking and unit checking aspects result in notable increases in runtime, with the type checking aspect running 2 times slower than the original program, and the unit checking aspect running 5.43 times slower than the original program. These slowdowns are the result of a large number of checks that must be made at every assignment to ensure the program is behaving as specified. In particular the fdtd benchmark runs 16.15 times slower when woven with units. The reason behind this is that there are many quantities which can be annotated with units, and as a result most computations require several secondary computations to ensure unit consistency. While it may seem cumbersome, one must consider that once a user has ensured their program is operating as expected, they can opt to cease usage of these aspects and their program returns to its original speed. The profiling aspects also feature a notable slowdown, running at 1.46 and 1.39 times slower than original program. However, much like the type and unit checking aspects, one would expect these aspects to only be used to find information about the program, and then returning to simply using the base MATLAB code. Given this usage, the slowdown is quite reasonable.

Unlike the other aspects, we hope that the loop unrolling and loop reversal aspects may lead to some performance increase, as their purpose is to perform some optimizations prior to running the program. While on average neither of these aspects outperforms the base

MATLAB code, we can note that the Loop Unrolling aspect runs faster for three benchmarks, crni, fdtd, and fiff, and that the Loop Reversal aspect runs faster on fiff. While using these aspects with all programs would not be advisable, as they do on average run 1.03 times slower, their utility lies in the fact that they make these program optimizations easy. It is straightforward to simply weave a program with these aspects, and if it performs better, continue using woven code or use it as an indication that one could consider performing the optimization manually, and disregard it if it results in a performance decrease.

6.3 Analysis of Engine Improvements

In this section, we demonstrate the value of the engine improvements made to the ASPECT-MATLAB compiler. First, we perform the same experiments as in the previous section, but without using the function inlining and the local copy of aspect variable optimizations. Of note, we do not perform this comparison with the loop optimization aspects, as they rely on these optimizations in order to function. The results of this experiment are shown in Table 6.2. As before, entries with a "-" indicate where the aspect did not apply. From these results, it is clear that the two optimizations have a very significant impact. The unit checking aspect runs 74.05 times slower than the base MATLAB program, making its use impractical. The type checking aspect, while not as bad, still performs 43.23 times slower, which again is an unreasonably large slowdown that makes it too cumbersome to use.

In Table 6.3, we show the results of the same experiment after enabling function inlining, but without the local copies of aspect variables. The significant decrease in runtimes that result from enabling function inlining demonstrate clearly the importance of that optimization. Clearly, the function call overhead in object-oriented MATLAB is important to consider for those who are concerned about performance, as eliminating it has obtained a speedup of about 5 times in the case of the unit checking aspect. As a lack of function inlining implies that the program must make function calls each time action code is to be executed, programs with a large number of pattern matches experience a much greater slowdown. Thus, aspects such as the type checking and unit checking aspects, which will often perform several actions for a single line of code, experience a much more significant slowdown than those aspects whose patterns match less frequently. Similarly, accesses to

	Time (Sec)	**Slowdown**			
Benchmark	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling
beul	24.95	25.71	50.51	14.85	13.73
crni	23.56	26.77	100.45	13.26	12.78
diff	23.82	24.18	39.51	13.04	12.41
fdtd	25.23	81.78	116.18	9.67	9.45
fiff	24.50	32.25	35.12	14.34	13.97
hnor	26.12	31.69	-	13.51	13.83
mils	25.05	68.67	-	12.48	10.74
nb1d	25.19	85.75	129.13	11.62	12.84
nb3d	24.67	64.33	115.58	9.89	11.66
capr	24.78	96.59	149.02	9.50	7.96
Geometric Mean		43.23	74.05	12.07	11.77

Table 6.2 Time in seconds for execution of benchmarks, and slowdowns with aspects woven without function inlining and without local copies of aspect variables

	Time (Sec)	**Slowdown**			
Benchmark	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling
beul	24.95	8.07	10.85	5.00	4.85
crni	23.56	8.07	14.34	5.25	4.67
diff	23.82	7.97	10.98	4.66	4.29
fdtd	25.23	9.94	22.25	5.52	5.39
fiff	24.50	6.24	9.88	5.71	5.29
hnor	26.12	4.23	-	4.49	3.89
mils	25.05	8.17	-	3.70	3.50
nb1d	25.19	5.63	16.68	5.74	5.65
nb3d	24.67	6.63	19.11	5.75	5.54
capr	24.78	4.99	16.00	6.13	5.99
Geometric Mean		7.02	14.26	5.14	4.84

Table 6.3 Time in seconds for execution of benchmarks and slowdown with aspects woven with function inlining but without local copies of aspect variables

aspect variables is time consuming, with all aspects performing about 3 times faster with the local copies being made. When using object-oriented MATLAB code, the time require to access object properties is significant, and any program which makes heavy use of object properties would likely see a notable performance increase through local copies.

6.4 Code size increase

It is worth noting that using aspects also results in a notable bloating of the program size after the action code has been woven. To evaluate the extent of this bloating, in table 6.4 we look at the lines of code (LOC) of each MATLAB benchmark, and note the increase in size for each aspect. All aspects result in a significant increase in code size, with the type aspect resulting in woven code that is on average 4 times longer than the original and the unit aspect resulting in woven code that is on average 14 times longer than the original. This is a significant increase, but it is to be expected, as most of the aspects weave multiple lines of code around many array accesses. In addition, many of the simplifications performed in order to allow for precise weaving also greatly expand the program size. For most applications, this significant increase in code size is likely not a concern.

	LOC	**Increase in LOC**					
Benchmark	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling	Loop Unrolling	Loop Reversal
beul	23	4.31	10.81	2.10	2.08	-	-
crni	194	3.89	12.15	2.24	2.26	42.12	3.49
diff	115	4.76	9.12	2.13	2.05	46.22	4.21
fdtd	77	5.20	43.19	1.98	2.13	37.28	-
fiff	105	3.42	16.84	2.38	2.34	31.81	4.12
hnor	22	3.14	-	2.27	2.00	44.65	2.46
mils	31	2.18	-	1.87	1.94	-	-
nb1d	166	4.95	12.36	2.34	2.37	57.16	-
nb3d	141	5.79	13.54	2.62	2.51	28.12	-
capr	206	4.30	12.07	2.56	2.24	15.23	-
Geometric Mean		4.03	14.66	2.24	2.19	40.06	3.49

Table 6.4 Size in lines of code of benchmarks and increase after weaving aspects

6.5 Summary

Overall, the type checking, unit checking, and profiling aspects discussed in chapter 4 do experience a notable slowdown, however, due to the ease with which aspects can be applied and removed, this presents no significant loss to those who would use these aspects to better understand their programs. The loop optimization aspects also perform slower on average, but again, due to the ease with which they can be applied and removed, make it easy to test the effects of optimizations. With the function inlining and local copy optimizations, ASPECTMATLAB code runs nearly 10 times faster than it did before - a significant improvement.

Chapter 7
Related Work

In this thesis, we present the development of an aspect-oriented compiler targeted at dynamically typed, array-based scientific languages. In this section, we contrast our work with other approaches to the problems we address.

There have been very few attempts at bringing aspect-oriented techniques to scientific programming. Of note is the work by Cardoso et Al. [CFM+13], who also target the MATLAB language, and bring aspect-oriented features to it. Their approach focuses primarily on the problems of monitoring variables and tracking behaviour in embedded systems. They implement their own aspect oriented language, which consists of a set of properties, analogous to ASPECTMATLAB patterns, which are then used in rules, which are analogous to ASPECTMATLAB actions to insert code into a separate base program. The implementation features only static pointcuts, matching function calls, variables, functions, tags, programs, and MATLAB reserved keywords. They introduce the concept of "tags", special comments that are acknowledged by their compiler in a similar fashion to ASPECTMATLAB annotations. These tags however, operate somewhat differently from our annotations, in that while they allow code to be woven around them, they cannot contain information which might be used by the woven code. In the rules demonstrated, these tags are primarily used to monitor when the compiler passes specific points in the base code.

Another approach to enriching scientific programming using aspect-oriented techniques is demonstrated by Irwin et al. [ILG+97]. They describe an approach to handling sparse matrix code through the use of aspect-oriented techniques. When dealing with sparse-matrix code, it is desirable to use a high-level matrix language (such as MATLAB), due to the ease with which you can prototype and produce easily understandable code. However, in doing so you forgo the advantages of using a lower-level language which can exploit appropriate data-structures to gain increased performance. Their approach allows for a user to write high level matrix code, but annotate it with information regarding an efficient implementation. Their approach to this dilemma was to create a language called AML, which allows for a user to write high level sparse matrix code, but annotate it with information regarding an efficient implementation. As a base language, they approximate the MATLAB language, but allow additional information to be passed regarding the data representation of matrices. This information is then interpreted by an aspect weaver, which weaves the appropriate data structure into the base code, which is then compiled into C/C++. Their approach, allowing for annotations in code to contain important implementation information is similar to what we accomplish with our annotate pattern. However, their work targets a very specific task, while our approach is far more broad.

A similar approach for handling the dynamic types in MATLAB using aspect-orientation, which inspired our current approach, has been documented in [Hen11]. It proposes the use of “atype" statements, which provide similar type information about variables as we require in our type annotations. The “atype" approach was a paper design and did not have an implementation. Another significant difference between this proposed strategy and the one we have adopted is the placement of annotations inside of MATLAB comments. The tradeoff here is that by placing type information in comments, it is possible to run annotated code without first using the ASPECTMATLAB compiler, however by using atype statements within MATLAB code, it would be possible to have an aspect which matches these statements, as well as a MATLAB function which executes the necessary type checking code even if aspect code is not woven - at some performance cost. For our implementation, we have decided that the best strategy would be to ensure that it is possible to run code without checks or performance loss.

Other approaches to types in dynamically typed scripting languages have been demonstrated by Ren et al. [RTSF13] , who introduce rtc, an annotation-based dynamic type checker for Ruby. Their design goals are similar to our own, in that they allow for checking types only when required, to support rapid prototyping and provide flexibility to the programmer. Their approach differs from our own in that it is provided in the form of a Ruby library with methods for type checking on objects, as opposed to using a separate compilation process.

In AspectJ, it is possible for patterns to match on constructs based on the annotations they are annotated with. Note, this is different from MATLAB, where our annotations do not annotate constructs, and instead designate important areas in code. Noguera et al. [NKDD10], introduce the idea of dynamic annotations to AspectJ. These achieve a similar goal to what we have done with ASPECTMATLAB annotations, though are presented in a very different context. Their implementation allows for annotations to communicate with the aspect compiler by associating them with conditions that determine when they should be active. This is somewhat limiting, as the communication is limited to whether the annotation is either in an on or off state, but succeeds at providing additional flexibility to users of AspectJ. One could imagine that a similar implementation of their dynamic annotations which take our approach to communication between base code and aspect code could increase its capabilities even further.

In our paper, we provide an aspect-oriented approach to dimensional analysis in MATLAB. Other approaches to handling dimensional analysis have been done as part of the Osprey project [JS06] by Jiang et al. They introduce a constraint-based approach which models units as types. They use annotations on variables, which contain similar information to our unit annotations, and use these annotations to statically detect and report errors. One advantage of our approach is that by operating at runtime, we can operate correctly in the face of several dynamic features endemic to MATLAB such as eval.

Chapter 8
Conclusions and Future Work

8.1 Conclusions

In this thesis, we have provided several valuable extensions to the AspectMatlab language. We have carried out various optimizations to improve the performance of aspect-woven code, and introduced several new aspects with the intent of helping Matlab programmers better understand and use their programs.

Our main goal was to make the AspectMatlab language more accessible to programmers who would be unfamiliar with aspect-oriented programming. We achieve this goal in several ways. First, through the introduction of multiple new patterns, we make it easier for programmers who use AspectMatlab to have their aspects function as desired. The introduction of the annotation pattern allows for programmers to communicate to aspect code through annotations, enabling an exchange of information that is relevant to aspect code. The type and dimension patterns allow programmers to further specify conditions under which they expect their action code to be woven. The new body keyword for around advice on loops allows more flexibility for programmers who wish to take advantage of loop patterns, which are core to AspectMatlab, given the loop heavy nature of scientific programming.

The introduction of new aspects makes it easy for programmers to get started using AspectMatlab. By introducing a variety of new aspects, it is possible for novice programmers to gain immediate benefits from using ASPECTMATLAB, even without a full understanding of the language itself. Our type and unit checking aspects allow for MATLAB users to better communicate the intentions of their code, and provide them with tools to verify that their code operates as expected. Our profiling aspects also extend programmers' ability to understand their code. The loop transformation aspects we have included make it easier for programmers to find ways in which they can optimize their code.

By implementing several optimizations to the woven ASPECTMATLAB code, we significantly improve the performance of aspect-oriented code. These optimizations have been demonstrated to have a significant impact, and expected performance gains have been shown to easily amount to an order of magnitude.

8.2 Future Work

In this section, we look into possible improvements that could be made to ASPECTMATLAB which build upon and address issues with our current implementation. It is our belief that as usage of ASPECTMATLAB grows, people will find new and exciting ways to use the tool. In this regard, the language holds much potential for extension, and can grow to suit the needs of its users with new patterns, targeting different MATLAB constructs. The development of more general-purpose scientific aspects would also contribute to the utility of ASPECTMATLAB. By packaging the compiler with a greater number of aspects that easily allow scientists to gain advantages with their MATLAB programming, there will be an enhanced incentive to use ASPECTMATLAB. Other areas of improvement lie in increasing the readability of woven code. Currently, woven code can be difficult for a human to parse, due in part to a number of simplifications that are necessary for pattern matching. These simplifications result in a low-level structure, which can obscure the meaning behind several operations. Transformations could be performed on the woven code to improve readability.

Performance improvement is another large area for future investigation. While our experience with inlining action code has shown it to be often beneficial, it is possible that a more selective heuristic for determining when function inlining should be performed could lead to some performance increases. Knowledge obtained from type annotations as we have described could also be used by a MATLAB JIT compiler to produce more efficient code. One could achieve performance gains by writing a MATLAB implementation which took annotations into account, perhaps by relying on the user ensuring type safety by running the program with our type checking aspect beforehand.

It is our hope that scientists will find use for the example cases provided, and use them as a platform to develop their own aspects, pushing the ASPECTMATLAB language in new and exciting directions.

Bibliography

[ADDH10] Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. AspectMatlab: An Aspect-Oriented Scientific Programming Language. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, March 2010, pages 181–192.

[Asl10] Toheed Aslam. AspectMatlab: An Aspect-Oriented Scientific Programming Language. Master's thesis, McGill University, 2010.

[CFM$^+$13] João M. P. Cardoso, João M. Fernandes, Miguel P. Monteiro, Tiago Carvalho, and Ricardo Nobre. Enriching matlab with aspect-oriented features for developing embedded systems. *J. Syst. Archit.*, 59(7):412–428, August 2013.

[Doh11] Jesse Doherty. McSAF: An Extensible Static Analysis Framework for the MATLAB Language. Master's thesis, McGill University, December 2011.

[Hen11] Laurie Hendren. Typing aspects for MATLAB. In *Proceedings of the Sixth Annual Workshop on Domain-specific Aspect Languages*, Porto de Galinhas, Brazil, 2011, DSAL '11, pages 13–18. ACM, New York, NY, USA.

[ILG$^+$97] John Irwin, Jean-Marc Loingtier, JohnR. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In Yutaka Ishikawa, RodneyR. Oldehoeft, JohnV.W. Reynders, and Marydell Tholburn, editors, *Scientific Computing* *in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 249–256. Springer Berlin Heidelberg, 1997.

[JS06] Lingxiao Jiang and Zhendong Su. Osprey: A practical type system for validating dimensional unit correctness of C programs. In *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2006, ICSE '06, pages 262–271. ACM, New York, NY, USA.

[Mat09] MathWorks. *MATLAB Programming Fundamentals*. The MathWorks, Inc., 2009.

[Mola] Cleve Moler. The Growth of MATLAB and The MathWorks over Two Decades. `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf`.

[Molb] Cleve Moler. The Origins of MATLAB. `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html`.

[NKDD10] Carlos Noguera, Andy Kellens, Dirk Deridder, and Theo D'Hondt. Tackling pointcut fragility with dynamic annotations. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, Maribor, Slovenia, 2010, RAM-SE '10, pages 1:1–1:6. ACM, New York, NY, USA.

[QMS98] M. J. Quinn, A. Malishevsky, and N. Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, 1998, HPDC '98, pages 114–. IEEE Computer Society, Washington, DC, USA.

[RP99] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[RTSF13] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The Ruby Type Checker. In *Proceedings of the 28th Annual ACM Symposium* *on Applied Computing*, Coimbra, Portugal, 2013, SAC '13, pages 1565–1572. ACM, New York, NY, USA.

Appendix A
Using ASPECTMATLAB

The current release of the ASPECTMATLAB compiler is can be downloaded from `http://www.sable.mcgill.ca/mclab/aspectmatlab/`

After obtaining a copy of compiler, it can be used to weave aspect files in one of two ways.

A.1 Execute Jar Directly

Among the included files, you can find and execute the ASPECTMATLAB jar directly. As an example, one may run `java -jar amc.jar aspect.m matlabfunction.m`, which would apply the aspect to the function. Any number of aspects and functions may be provided, and each aspect will be woven to each function. A `weaved` directory will be placed in the current working direction, and code woven by the compiler can be found within.

When running from a terminal, ASPECTMATLAB allows for several flags, outlined below.

-main A MATLAB function file can be specified as the entry point to a program by inserting the `-main` flag before the function name.

-m By default, standard MATLAB code is translated into Natlab code prior to weaving. Using the `-m` tag skips this translation.

-out The output directory can be specified using a `-out` flag, followed by the directory name.

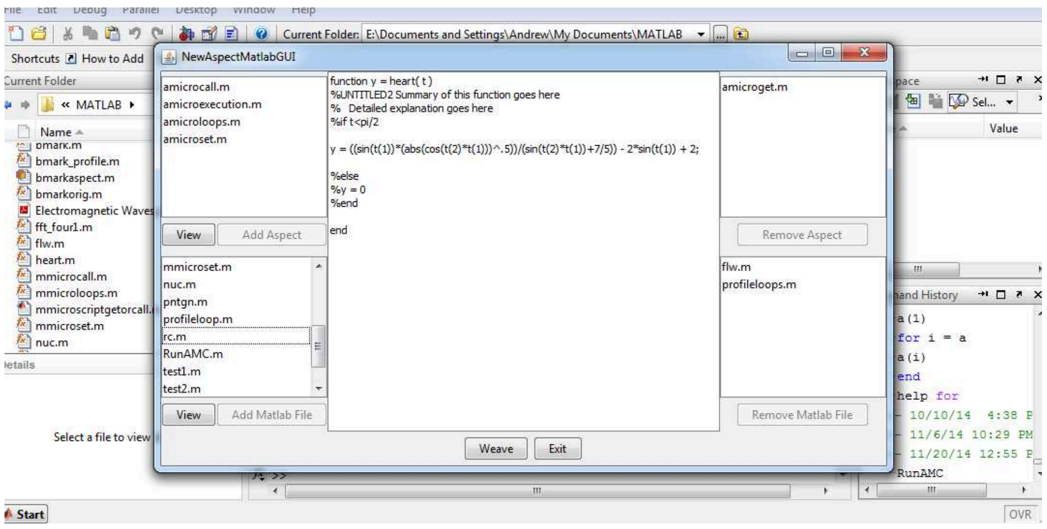
-version The version number can be checked with the `-version` flag.

-help The `-help` or `-h` flag can be can be used to describe useage of the ASPECTMATLAB compiler.

A.2 Using ASPECTMATLAB from within a MATLAB environment

To make ASPECTMATLAB easier to use, we have included in this release an interface that can be used from within a MATLAB environment. This interface can be used to choose aspect files and MATLAB functions to be woven, and allows for weaving with the push of a button. To use ASPECTMATLAB within MATLAB, simply place the `amc` directory into the working directory of your MATLAB environment. Then, simply call the `runAMC` function. The interface shown in *Figure A.1* will be displayed.

Figure A.1 Using ASPECTMATLAB in a MATLAB environment.



```
function y = heart( t )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
%if t<pi/2

y = ((sin(t(1))*(abs(cos(t(2)*t(1)))^.5))/(sin(t(2)*t(1))+7/5)) - 2*sin(t(1)) + 2;

%else
%y = 0
%end

end
```

To add aspects, select the desired aspect from the top left box, and press the "Add Aspect" button. Added aspects will be displayed in the top right box, and can be removed with the "Remove Aspect" button. To add a MATLAB file, select the desired MATLAB function in the bottom left box and press the "Add Matlab File". Added MATLAB functions will be displayed on the bottom right, and can be removed with the "Remove Matlab File" button. The "View" buttons can be used to preview aspects and MATLAB files, displaying their contents in the center pane. Once all desired files have been selected, press the "Weave" button to run the ASPECTMATLAB compiler with the selected aspects and MATLAB files. The woven output will be placed in a `weaved` directory.