McGill University
School of Computer Science
Sable Research Group

AspectMatlab: An Aspect-Oriented Scientific Programming Language

(Extended version)

A shorter version of this paper appears in AOSD 2010, please cite the AOSD paper

Sable Technical Report No. sable-2009-03

Toheed Aslam, Jesse Doherty, Anton Dubrau and Laurie Hendren

Revised January 15, 2010

Contents

List of Figures

List of Tables

Abstract

This paper introduces a new aspect-oriented programming language, AspectMatlab. MATLAB® is a dynamic scientific programming language that is commonly used by scientists because of its convenient and high-level syntax for arrays, the fact that type declarations are not required, and the availability of a rich set of application libraries.

AspectMatlab introduces key aspect-oriented features in a way that is both accessible to scientists and where the aspect-oriented features concentrate on array accesses and loops, the core computation elements in scientific programs.

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

This paper reports on the language design of AspectMatlab, the *amc* compiler implementation and related optimizations, and also provides an overview of use cases that are specific to scientific programming.

1 Introduction

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

Our challenge was to define and implement a new aspect-oriented programming language that was a natural extension of MATLAB. We wanted to build upon the successes of languages such as AspectJ [2,9], but at the same time tailor our approach to the needs of the scientific programmer. In particular, we wanted to introduce new language features for matching array and loop operations, both of which are central to scientific programming. We also wanted to introduce aspect-oriented programming in a way that was a natural extension to the MATLAB language and so that it would be understood and adopted by the scientific programmers.

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

AspectMatlab supports traditional patterns (pointcuts) such as `call` and `execution`, but we have also concentrated on an effective design for `get` and `set` patterns which naturally deal with arrays. Loops are key control structures in scientific programs and we have developed a collection of patterns which allow one to match on loops in a variety of ways. We have also been inspired by AspectCobol [11] in that we expose join point context information via selectors that are associated with actions.

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

We have implemented the *amc* compiler which translates AspectMatlab source files to pure MATLAB source files. The generated code can be run using any MATLAB system. The overall structure of

the compiler was inspired from the abc [1, 3] system and is built as an extension of the McLab[1] MATLAB front-end. In implementing the compiler it became clear to us that weaving into MATLAB code offers new challenges that are different from weaving into more statically-typed, traditional languages such as Java. As one example, the expression `a(i)` may be either a call to function `a` or a get of the `i`'th element of array `a`. Even worse, the precise rules for looking up names differs for functions, inner functions and scripts. Thus, a naive weaving strategy for MATLAB requires a lot of dynamic checks to determine if an expression matches.

To deal with the special challenges of weaving in MATLAB, we have implemented some intra-procedural flow analyses using our new McLab analysis framework which enables us to statically determine whether names correspond to variables or functions. Applying these analyses before weaving allows us to greatly reduce the number of dynamic checks required.

The main contributions of this paper are: (1) the AspectMatlab language definition (Section 2) and compiler (Section 4); (2) new scientific patterns (pointcuts) and use cases (Section 3); and (3) optimizations to reduce dynamic checks (Section 5). We also provide a discussion of related work in Section 6 and give conclusions and future work in Section 7.

2 AspectMatlab Language

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

In AspectMatlab, aspects are defined using a syntax similar to MATLAB classes. A MATLAB class typically contains properties, methods and events. So taking advantage of the class structure, an aspect retains the properties and methods, while adding two aspect-related constructs: patterns and actions. Patterns are formally known as pointcuts in AspectJ and of course, AspectMatlab actions correspond to AspectJ advice. This choice of terminology was intended to convey that patterns specify where to match and actions specify what to do. An example of a complete AspectMatlab aspect is given in Figure 1.

2.1 Patterns and Actions

Just like any other aspect-oriented language, AspectMatlab provides a variety of patterns that can be used to match basic language constructs. In addition to standard patterns such as those supported by AspectJ, a scientific programming language like MATLAB possesses other important cross-cutting concerns. In MATLAB, array constructs are heavily used and programs are written in the form of large functions or scripts containing many loops. While providing the basic function-related patterns like `call` and `execution`, we also introduce two new sets of patterns: (1) get/set patterns, enabling the facility to capture the array-related operation along with useful context exposure; and (2) loop patterns, which will help programmers to handle the loop iteration space

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

and details of loop-intensive computation. In addition to that, we also provide the `within` pattern to restrict the scope of matching to certain named constructs of the program, such as functions, scripts, classes or loops.

An example of a simple `call` pattern and a corresponding named action is shown below. `pCallFoo` matches all the join-points only within the function `bar`. Action `aCountCall` is invoked before all such join points in the source code.

```
patterns
  pCallFoo : call(foo) & within(function, bar);
end

actions
  aCountCall : before pCallFoo
    %action code
  end
end
```

We present examples of function patterns in section 2.4, while the rest of the patterns are explained in detail in Section 3. A list of patterns supported by AspectMatlab is given in Table I.

Table I: List of Patterns

call	captures all calls to functions/scripts
execution	captures the execution of function bodies
get	captures array accesses
set	captures array sets
loop	captures execution of all loops
loophead	captures the header of the loop
loopbody	captures the body of the loop
within	restricts the scope of matching

An aspect can contain many actions, and as in other aspect-oriented languages, there are `before`, `around` and `after` actions. Since multiple actions can be triggered at the same join point, if more than one of such actions are of the same type, we need default precedence rules for the actions:

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

An important point to notice here is that the default ordering rules of AspectMatlab are simpler and more restrictive than the precedence rules of AspectJ [9]. However, our action weaving strategy

avoids complicated dependency rules, will not lead to any dependency cycles between actions, and is easy to comprehend from a scientific programmer's point of view. Since our actions have names, it would also be simple for us to introduce a declaration to over-ride the default ordering within each of the around, before and after groups.

2.2 Selective matching

As shown in the example below, pattern `call2args` will match all calls, but only the ones made with two or more arguments. This enriched pattern syntax allows selective matching, because in MATLAB a function call does not necessarily have to provide exactly as many arguments as specified in a function signature.

```
patterns
  call2args : call(*(*, .. ));
  % other patterns
end
```

Also, the syntax for accessing the arrays and cellarrays is the same as of calling a function. So this selective matching is available on a range of patterns, which we shall describe in the following sections.

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

Table II: Selective Pattern Matching

call(foo)	matches all calls to foo
call(foo())	matches calls with no arguments
call(foo(*))	matches calls with exactly one argument
call(foo(..))	matches calls with 1 or more arguments
call(foo(*,..))	matches calls with 2 or more arguments
	...and so on

2.3 Context exposure

When it comes to the context capture, AspectCobol's [11] design doesn't rely on the use of reflection inside the advice code, as performed in AspectJ [9]. Rather, it suggests that join point reflection on the static shadow should be a part of the pointcut. The extraction of the context-specific information is described as part of the pointcut designator. We extend the idea of binding the results of desired context variables for subsequent use in the action code.

In AspectMatlab, access to the static program context that belongs to the join point is selector based. These selectors are specified along with an action definition, because an action corresponds directly to the static join point shadow. In the example below, the action `actcall` will fetch the `name` and `args` of the function call from the join point shadow.

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

Of course, a selector is only applicable depending upon the join point type. For example, the `counter` selector is only meaningful when used on a loop join point. A list of context selectors and their meaning with different join points is given in Table III.

Table III: Context Selectors for different join point types

	get	set	execution	call	loop	loopbody	loophead
args	indices		arguments passed		-	-	-
obj	variable	variable before set	-	function handle	loop iterator variable		
newVal	-	new array	-	-	-	-	range expression
counter	-	-	-	-	-	current iteration	-
line	line number in the source code						
loc	enclosing function/script name						
name	name of the entity matched				-	-	-
varargout	cell array variable used to return data from `around` action						

In Figure 1, we present an example of an aspect, which counts all the function calls made with at least two arguments. To do so, we need to have a `call` pattern to capture all such calls. The `execution` pattern is used to display the number of calls made at the end of the program.[2]

2.4 Small Example

To demonstrate the application of the aspect from Figure 1, consider a small base program consisting of the simple MATLAB function given in Figure 2.

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

Once compiled along with the aspect presented in Figure 1, pattern `call2args` finds only three matching join points (at lines 5, 6 and 9) where the function calls carry two arguments each. So, corresponding action function calls will be woven only at those program points. Note that the function calls with a single input argument (at lines 11, 12 and 13) do not match. Moreover, the action `actexecution` is an `after` action, so it will be woven at the end of the function. The woven code generated by the compiler is shown in Figure 10. It will be easier to follow the output after we explain how different phases of the compiler work in Section 4.

[2]Note that we defined the `actcall` action as an **around** action, but it could also be defined as a **before** action.

```
aspect myAspect

properties
  count=0;
end

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

patterns
  call2args : call(*(*, .. ));
  executionMain : execution(histo);
end

actions
  actcall : around call2args : (name, args)
    this.incCount();
    disp(['calling ', name, 'with parameters(', args , ')']);
    proceed();
  end
  actexecution : after executionMain
    count = this.getCount();
    disp(['total calls : ', num2str(count)]);
  end
end %actions

end %myAspect
```

Figure 1: An aspect to count all calls made with at least 2 arguments

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

Figure 2: MATLAB function

3 Scientific Patterns and Use Cases

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

All examples can be found online at `http://sable.mcgill.ca/mclab/aspectmatlab/examples`. They all include the aspects and the programs that are modified, as well as woven code generated by *amc* (i.e., the compiler is not needed to check the benchmarks). In general, we consider two possible use cases: (1) profiling programs, and (2) annotating data to variables in a running program to extend functionality.

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

With regards to annotating functionality it is interesting to note that the McLab Project was conceived as a framework not only to allow the addition of analyses and compilation of Matlab into different backends. It is also a framework to allow the simple development of language extensions, which is exactly what the *amc* compiler represents (an extension of the base MATLAB compiler). Aspects are a quick way to prototype further possible language extensions without much work, as the units (Section 3.3) benchmark shows.

3.1 Tracking array sparsity

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

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

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

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

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

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

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

Figure 3: Outline of sparsity aspect

shape as well as sparsity changes.

The existence of the get and set patterns are particularly convenient here, because we merely have to write actions in which we increase counters associated with every variable. Since the context information includes the name of a matched variable as a string, we can put all the variables in a MATLAB structure to map between names and values. A structure in MATLAB, unlike in static programming languages, allows the addition of fields during runtime. As new variables are encountered during runtime, they are added into the structure that tracks them, so we don't have to specify the variable names in advance. Thus, the aspect needs almost no modification to profile different programs. In fact the only modification needed is the pattern that specifies the call of the main entry point of a program, because MATLAB does not have a standard `main` entry point.

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

Along with the aspect itself we coded our actual program, which utilizes a RungeKutta4 ODE solver [12] to solve the heat equation in 1D given some initial conditions and time interval. The benchmark uses matrices to discretize the heat function in space. The needed derivative is computed using matrix multiplication with a differentiation matrix which is very sparse and never changes. Most of the computation of the program relies on this multiplication. If this matrix is made sparse, it decreases the overall computation time for this benchmark by 95% (tested in Matlab R2008a, on a linux PC with an AMD Athlon 64 X2 with 2GHz and 4GB of ram).

The output of the benchmark in Figure 4 clearly shows that the variable D is of large size, never indexed, seldomly written or changing in shape or sparsity, but often used without indexes. We

thus show a very simple benchmark using aspects and the special array patterns to profile a certain feature of a program, leading to a useful result. Without aspects and these patterns, one would need to inline profiling code manually.

```
>> program
tracking sparsities of all variables in the following program...
computation finished
  'var'   'size'              'sparsity'    'arraySet' 'shape changes' 'decrease sparsity' 'increase sparsity' 'get' 'indexed get'
  'a'     '1.0 +-0.0'         '1.00 +-0.04'  [     2]   [          2]   [              1]   [              0] [2002] [          0]
  'h'     '1.0 +-0.0'         '1.00 +-0.00'  [     2]   [          2]   [              0]   [              0] [3504] [          0]
  'X'     '299.0 +-0.0'       '1.00 +-0.00'  [     1]   [          1]   [              0]   [              0] [   1] [          0]
  'U0'    '199.7 +-140.5'     '0.37 +-0.45'  [     2]   [          2]   [              1]   [              1] [   1] [          0]
  'D'     '89356.4 +-1997.0'  '0.01 +-0.02'  [     3]   [          3]   [              2]   [              1] [2000] [          0]
  'W'     '124874.0 +-55722.8' '0.55 +-0.33'  [   504]   [        504]   [              2]   [            501] [   1] [       2500]
  't'     '251.1 +-250.0'     '1.00 +-0.02'  [  2001]   [       2001]   [              2]   [              0] [   0] [       2000]
  'u'     '299.0 +-0.0'       '0.97 +-0.13'  [  2000]   [       2000]   [            135]   [              0] [2000] [          0]
  'k1'    '299.0 +-0.0'       '0.97 +-0.13'  [   500]   [          1]   [              1]   [             34] [1000] [          0]
```

Figure 4: Output of the sparsity benchmark

3.2 Measuring floating point operations

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

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

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

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

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

We used this aspect to weave into the computation of the singular value decomposition [13] of a random matrix. The utilized algorithm is spread over many files and operates on many 2x2 submatrices as well as the whole matrix, and it is not clear which operations dominate.

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

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

    any     : call (*);
  end % patterns

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

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

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

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

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

Figure 5: Outline of flops aspect

As the output in Figure 6 shows, the aspect is able to uncover where most of the computation happens, and presents it in a similar way a profiler shows computation time (i.e., encapsulated information). In the listing, lines with 0 flops were removed, as well as several very similarly behaving scalar operations (the line above the '...'s is representative of the omitted lines).

While it would be possible in MATLAB to override the behaviour of plus, minus etc (i.e., the atomic functions for which the aspect tracks the flops) to track the number of operations, it would be pretty much impossible to get that information in the way it is listed, i.e., with a report for every call site, and with encapsulated information, without emulating the before and after actions in some way. Moreover, the aspect allows the tracking of any function (which uses the tracked atomic operations) without modification, and only slight modification of the benchmark - merely

the function that should be tracked has to be edited.

```
>> runsvd
encountered call to SVD, recording flops...
finished tracking function call, here are the results:
   'call site'                   '# of calls' 'total flops'
   'fro_150_times'               [         1] [        100]
   'fro_150_sqrt'                [         1] [          1]
   'SVD_13_fro'                  [         1] [        101]
   'SVD_14_abs'                  [         7] [        630]
   'tinySVD_77_minus'            [       270] [        270]
   ...
   'tinySVD_81_mtimes'           [       270] [       3240]
   'tinySymmetricSVD_109_minus'  [       270] [        270]
   ...
   'tinySymmetricSVD_113_times'  [       270] [        270]
   'tinySymmetricSVD_116_mtimes' [       540] [       6480]
   'fixSVD_137_mtimes'           [       270] [       3240]
   'fixSVD_138_mtimes'           [       270] [       3240]
   'fixSVD_141_mtimes'           [        11] [        132]
   'fixSVD_142_mtimes'           [        22] [        264]
   'fixSVD_143_mtimes'           [        11] [        132]
   'tinySymmetricSVD_121_fixSVD' [       270] [       7008]
   'tinySVD_82_tinySymmetricSVD' [       270] [      17268]
   'tinySVD_83_mtimes'           [       270] [       3240]
   'jacobi_42_tinySVD'           [       270] [      25368]
   'SVD_17_jacobi'               [       270] [      25368]
   'SVD_18_mtimes'               [       540] [    1026000]
   'SVD_19_mtimes'               [       270] [     513000]
   'SVD_20_mtimes'               [       270] [     513000]
   'SVD_34_times'                [         1] [        100]
   'Script_6_SVD'                [         1] [    2078199]
```

Figure 6: Output of the flops benchmark

3.3 Adding units to computations

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

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

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

All operations (again only the functions, not the operators) are overridden to both perform the denoted operation on the .val field and the .unit field. Units are stored as vectors, denoting the power of every SI unit. There are 7 SI units, and they are ordered as metre, kg, second, Ampere,

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

Figure 7: Outline of units aspect

Kelvin, candela and mol. Thus, [1 0 -2 0 0 0 0] would denote m/s^2. The function 'dis' is overridden as well to show the matrix with the associated unit.

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

```
for i = x
```

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

In Figure 7, the action takes the range expression, and iterates over the values. These are stored in a cell array, which is then passed to the struct function which creates a structure array. This is a feature of MATLAB- when 'struct' receives a cell array, it will build a struct-array. When looping over this new structure, every element will be a structure containing the elements of value of the previous array.

For example, one could run (as in the provided example):

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

For which the result after weaving and running would be

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

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

3.4 Other possibilities

There are many more possibilities for aspects in the scientific computing domain utilizing our special patterns. For example, one could use the loop patterns to track an iterative solver like a Newton solver or a Jacobi solver [12]. This could be particularly useful if it is used inside some larger computation like a backward Euler integration [12], because it would allow one to see how many iterations are done when and where.

Tracking loop counts could also be interesting for loop dependency analyses. One could use aspects to collect runtime information and feed that back to the compiler to write specializing code optimizing the encountered runtime properties. Aspects could help with profile-guided optimizations in general, because they can simplify writing ad-hoc profilers for every desired optimization.

4 AspectMatlab Compiler

The AspectMatlab compiler has been designed to be easily extensible so that it is simple for us and other researchers to add further features. To enable this we have built it using extensible toolkits and have aimed for a very clean and modular implementation. In this section, we briefly describe the structure of the AspectMatlab compiler and its various phases.

The overall structure of *amc*, the AspectMatlab compiler is given in Figure 8. The compiler takes as input, a collection of MATLAB (.m) source files, plus a collection of AspectMatlab files, and produces a collection of woven MATLAB source files. These output files can be executed using any MATLAB system.

The front-end of AspectMatlab was implemented as an extension to the Natlab front-end (Natlab is a "neat" version of MATLAB, developed by the Sable Research Group). The scanner is built using the MetaLexer tool [4] and was specified as a simple and modular extension to the Natlab Metalexer specification. The parser and semantic checks were modular extensions to Natlab's parser, which is built using the extensible JastAdd framework [6]. The Natlab grammar was extended to incorporate AspectMatlab grammar rules using the JastAddParser. JastAdd provides powerful facilities for AST traversal, associating attributes with nodes and modifying the AST via node rewriting.

As indicated in Figure 8, after front-end processing, the AST generated includes both MATLAB and aspect-specific AST nodes. Following the *abc* model, the Separator component harvests all the aspect-specific key information out of the AST, and transforms the AST so that it becomes a pure MATLAB AST. This process allows us to process the resulting AST using our Matlab compiler analysis framework, and is also the first key step in converting the aspect source files to MATLAB source files. The separation phase records the aspect information into corresponding data structures

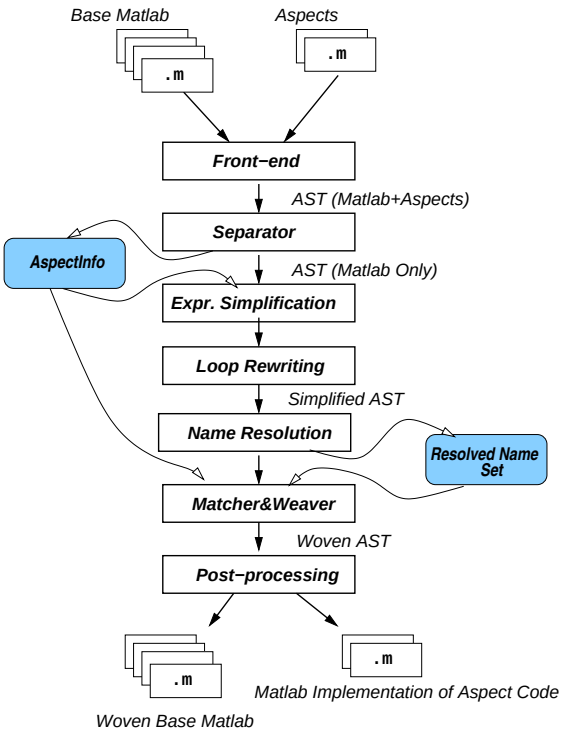

Figure 8: Overall structure of the *amc* AspectMatlab compiler

set, called AspectInfo (following the structure of the *abc* compiler). AspectInfo contains the pattern lists, action lists and the information about their association.

4.1 Transformations

In order to perform matching and later weaving, some join points require transformation. There are two notable code transformations: name/parameterized expressions simplification and loops rewriting. An expression in MATLAB can be very complex with a lot of computation being performed within a single expression. This computation can be in the form of function/script calls or complex operations on arrays. So, some kind of refactoring of complex expressions is required to expose all the matching and weaving points in the code. To avoid inserting meaningless and redundant code, we consult AspectInfo data structures at this stage. All the name or parameterized expressions, which potentially match the specified patterns, are taken out of the parent expression. This results in simple weavable statements with precise locations for before, after or around actions.

For example, given the following base program.

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

Assuming that patterns exist for both calls and variables, the above line gets translated into this:

```
AM_CVar_5 = x;
AM_CVar_6 = sum(AM_CVar_5);
AM_CVar_7 = y;
AM_CVar_8 = length(AM_CVar_7);
```

```
z = (AM_CVar_6 / AM_CVar_8);
```

It should be noted that arguments of the function calls are extracted out for the sake of their own weaving. In turn, in the `around` case, if the function never gets called through `proceed`, its arguments were still evaluated before passed on to the `around` action.

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

For example, consider the following loop:

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

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

```
AM_CVar_1 = 1:step:size(dx,1);
for AM_CVar_2 = 1:numel(AM_CVar_1)
  i = AM_CVar_1(AM_CVar_2);
  % loop body
end
```

`while` loops present a different challenge. The conditional expression can contain several instructions inside it. Refactoring the expression will be our solution again. But since the condition is supposed to be evaluated at the start of each iteration, we have to take care of all the back edges of the loop finishing at the loop header, which means just before the syntactical end of loop body and also at all the `continue` statements.

For example, consider the following loop.

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

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

```
AM_CVar_3 = x < y;
while AM_CVar_3
  % ...
  AM_CVar_3 = x < y;
  continue;
  % ...
  % loop body
```

```
  % ...
  AM_CVar_3 = x < y;
end
```

4.2 Name Resolution Analysis

In MATLAB, a function call or an array access has the same syntax using either just the name of the entity or passing a number of parameters with it. So `foo(1, 2)` can either be an access to an array named `foo`, if it exists in the current scope, or it could be a function call with two parameters. This name resolution can be achieved with the help of runtime checks, but doing so we compromise on the efficiency of generated weaved code. So we essentially need to have a flow analysis, to determine the exact type of a join point at the time of matching. We describe this analysis in detail in section 5.

4.3 Matching and Weaving

The previous name resolution phase populates the resolved names set, which is then used as one input to the matcher and weaver.

Matching of the patterns and then weaving the action advice happens in a single pass through the AST. Before/after `execution` actions are woven by simply inserting an action call at the start or at the end of the function, respectively. In case of around of `execution` (and other kinds of patterns as well), the semantics of MATLAB force us to develop a different strategy, which is described in section 4.3.1.

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

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

The lowest tier of matching is at the statement level. Since we have simplified the complex statements already in an earlier pass, it only comes down to assignment statements or even simple expression statements. The left hand side of an assignment statement is matched for `set` patterns, and right hand side expression for `get` or `call` patterns. This is where we can use the name resolution set, which helps us determine if the expression is a `get` or `call` join point. Without the name resolution optimizations we must weave in a dynamic check.

```
function [varargout] = myAspect_actcall(this, AM_caseNum,
                AM_obj, AM_args, name, args)
  % Not all input context variables are shown for brevity.
  % All contaxt info is passed in case of multiple arounds.

  this.incCount();
  disp(['calling ', name, 'with parameters(', args, ')']);
  proceed(AM_caseNum, AM_obj, AM_args);
  function [] = proceed(AM_caseNum, AM_obj, AM_args)
  switch AM_caseNum
    case 0
    varargout{1} = AM_obj(AM_args{1}, AM_args{2});
    ... % other cases
    case 2
    AM_obj(AM_args{1}, AM_args{2});
  end
  end
end
```

Figure 9: Example of an `around` function

4.3.1 Weaving `around` actions

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

The solution we came up with is partially inspired by Kuzins' work on efficient implementation of around advice for the AspectBench Compiler [10]. Taking advantage of the MATLAB's nested functions, we create a nested function, namely `proceed()`, inside the around action function. This function contains a `switch` statement to host the extracted code from all the around join points of this particular action. The join points are assigned a simple number id, which is unique for each around action. Along with this unique id, a join point has to pass the context information to execute the extracted code being moved inside a different scope.

The `around` action from the example given in Section 2.4 is given in Figure 9.

4.4 Post-processing and Example

At the end of the weaving, a post-processing phase takes place. In all functions and scripts, we weave in the global structure which contains the aspect class objects and also few checks are inserted to validate the objects. Finally, *amc* generates standard MATLAB code.

After this detailed description of all phases, we come back to the example given in Section 2.4. The woven code is shown in Figure 10. Expression simplification is very noticeable, as we transform complex statements into easy-to-weave statements. With the help of our name resolution analysis, we are able to distinguish between function calls and array accesses. All the calls matching the

```
function [m, s, d] = histo_am(n)
  global AM_GLOBAL;
  if (¬isempty(AM_GLOBAL))
    AM_EntryPoint_0 = 0;
  else
    AM_EntryPoint_0 = 1;
  end
  if (¬ isfield (AM_GLOBAL, 'myAspect'))
    AM_GLOBAL.myAspect = myAspect;
  end
  AM_CVar_0 = n;
  AM_CVar_1 = AM_GLOBAL.myAspect.myAspect_actcall(
      0, @randn, {AM_CVar_0, 1}, 'randn', {AM_CVar_0, 1});
  % Generate vectors of random inputs
  % x1 = Normal distribution N(mean=100,sd=5)
  % x2 = Uniform distribution U(a=5,b=15)
  x1 = ((AM_CVar_1 * 5) + 100);
  AM_CVar_2 = n;
  AM_CVar_3 = AM_GLOBAL.myAspect.myAspect_actcall(
      1, @rand, {AM_CVar_2, 1}, 'rand', {AM_CVar_2, 1});
  x2 = (5 + (AM_CVar_3 * (15 - 5)));
  AM_CVar_4 = x2;
  AM_CVar_5 = x1;
  y = ((AM_CVar_4 .^ 2) ./ AM_CVar_5);
  AM_CVar_6 = y;
  AM_GLOBAL.myAspect.myAspect_actcall(
      2, @hist, {AM_CVar_6, 50}, 'hist', {AM_CVar_6, 50});
  AM_CVar_7 = y;
  AM_CVar_8 = mean(AM_CVar_7);
  % Calculate summary statistic
  m = AM_CVar_8;
  AM_CVar_9 = y;
  AM_CVar_10 = std(AM_CVar_9);
  s = AM_CVar_10;
  AM_CVar_11 = y;
  AM_CVar_12 = median(AM_CVar_11);
  d = AM_CVar_12;
  AM_GLOBAL.myAspect.myAspect_actexecution();
  if AM_EntryPoint_0
    AM_GLOBAL = [];
  end
end
```

Figure 10: Woven MATLAB function

`call` pattern are woven accordingly. Notice the extra bit of code added by the post-processing phase, at the start and the end of the function.

5 Name Resolution Analysis

In this section we describe a program analysis needed to drive a weaving optimization. We begin by describing the motivation for this optimization, followed by a description of the analysis, and finally a discussion of the accuracy of the analysis. This analysis has been implemented using the McLab Analysis Framework and incorporated into the AspectMatlab compiler.

5.1 Motivation

To weave `call`, `get`, and `set` actions correctly, the compiler must know if a given name is a function or a variable at a given program point. If the name is a variable, then the compiler must know if the variable is defined at that point. As was described previously, MATLAB does not syntactically differentiate between function calls and variable reads. In addition, MATLAB does not predefine or initialize a variable before its first assignment. These properties of MATLAB mean that the information needed is not readily available to the compiler.

It is possible to naively weave these actions without having access to this information. To do this, the compiler must introduce runtime checks. In the case of determining if a name is a variable or a function, this check would look like the following example which has been shortened for brevity:

```
if (exist('x', 'var') ≠ 1)
  AM_CVar_0 = AM_GLOBAL.act_call(0, eval('@x'), {});
else
  AM_CVar_0 = x;
end
```

Not only do these checks incur a performance cost, they also introduce the use of `eval`. Introducing an `eval` is not desirable because it makes it difficult to perform subsequent analysis on the generated code.

Ideally we would like to eliminate many, if not all of these runtime checks and uses of `eval`. The analysis that will be described next is able to provide much of this information, and in some situations eliminate the use of `eval` entirely.

5.2 Analysis

The goal of this analysis is to determine if a given name at a given program point corresponds to a function, variable or assigned variable. To accomplish this goal the analysis is implemented as a data flow analysis using the McLab Analysis Framework. In this section we describe some of MATLAB's semantics relevant to this analysis, and follow with a description of the analysis implementation.

5.2.1 Matlab semantics

MATLAB's semantics offer novel challenges to weaving that are not present in languages with stronger static semantics such as Java. Since MATLAB is not compiled, many of the semantics are either runtime semantics or static semantics applied when the code is first loaded. We will refer to the latter as load-time semantics. However, these load-time semantics are not purely static.

They depend on the runtime state at the point when the code is loaded. Even worse, MATLAB's semantics become substantially different depending on if the code being executed is a script, a function, or a function containing inner functions. We begin by describing these three execution environments, starting with functions.

Functions in MATLAB behave as one might expect. They are callable. They take input parameters and have return parameters. They also introduce a new scope. In functions, MATLAB applies load-time semantics to determine if a given identifier is a variable or function, which guides name lookup at runtime. It determines this based on the first use of that name. If it is assigned a value first, then it is a variable. If it is read from first and it corresponds to an available function at load time, then it is a function.

Inner functions and functions with inner functions behave similarly to functions without inner functions. One main difference is that inner functions can access the scope of their parent function.

Scripts are similar to functions in that they are callable. However, unlike functions, they do not have parameters and do not introduce scope. Scripts are executed entirely within the scope of the calling context. What this means is that anything that was a variable in the calling context is a variable in the script and everything that was a function in the calling context starts off as a function in the script. A name can start off as a function but it can still be assigned a value. After that assignment, the name will be treated as a variable. However, this change is restricted to the script environment. It will only affect a calling environment if the caller was a script as well. So unlike functions, scripts rely more heavily on runtime semantics to perform a name lookup. There are further details concerning how the calling context is affected by script calls from within scripts, both in the callee and the caller, but we will not go into these details here.

These semantics mean the compiler must estimate what a name corresponds to at runtime. It must also estimate when a variable is known to have a value. These estimations are implemented as a static flow analysis in the McLab Analysis Framework.

5.2.2 Analysis implementation

The Name Resolution Analysis is designed to solve the following problem:
Problem: For every program point p and every name n, determine if n is a function, a variable, or an assigned variable at p.

This is implemented by a flow analysis that builds up a set of information for each statement in the program. These sets contain names labeled with the information about that name so far. This information is represented as one of five tags. These tags are described briefly in Table IV.

Table IV: Flow data values

$\top$	error
`AVAR`	assigned variable
`VAR`	variable with no assignment information
`FUN`	function
$\bot$	no information

The tags are organized in a lattice structure. This lattice is depicted in Figure 11. The reason the tags are organized in this lattice is to facilitate the flow analysis.

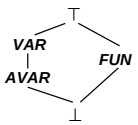

Figure 11: Abstraction Ordering

When a name n is labeled with a given tag at a particular program point p it means that for all possible execution paths to p, n was used in a way consistent with that tag. In particular this means that if n is labeled `AVAR`, then you are guaranteed that at that point n is a variable with a value. If n is labeled `VAR`, then n is guaranteed to be a variable, but it is unknown if it has been given a value. If n has the label `FUN`, then it must be a function. If n is $\bot$ or it is not labeled, then no information has been acquired about n up to this point. If n is $\top$ then there is at least one point on an execution path on which n was guaranteed to be a variable, and another point on an execution path on which n must have been a function.

The Name Resolution Analysis builds up this label information in two phases. The first phase is only applied to functions and is intended to estimate the load-time semantics. The second is applied to both.

The first phase estimates the load-time semantics by performing a preorder traversal of a function's abstract syntax tree (AST) and applying rules when encountering names. The result of these rules is a set of labeled names where the labels are one of the tags in Table IV. There is only one set for the entire function. The reason for only having one set is that the MATLAB semantics mean a name in a function cannot change from a variable to a function or a function to a variable. This means that if a name is determined to be a function or a variable, then it must be so for the entire function.

The rules that are applied are fairly simple and are as follows:

- The input or return parameter names are labeled `VAR`.
- All names on the left hand side of an assignment are labeled `VAR`.
- All names declared to be global or persistent are labeled `VAR`.
- All names that are used with the `@` operator to create function handles are labeled `FUN`.

If the analysis attempts to label a name with `VAR` or `FUN`, but the name was already labeled with the other, the name to be labeled with $\top$. If the analysis tries to label a name that is already $\top$, then there is no change. This phase is strong enough to be able to label all variable names as variables and it can be assumed that all other names are functions. Unfortunately this phase is too simple to get any information about assignments to variable.

The second phase estimates the runtime semantics. It does so by performing a forward flow analysis over the function or script. During this process it associates a set of labels with each statement.

In order for this analysis to start it must assume a starting set of labels. In the script case it uses the empty set. In the function case it uses the result of the first phase of the analysis.

Once a starting set is chosen, a simple structure-based data flow analysis is performed. The analysis applies rules to certain names, handles different execution paths by merging the results when paths converge, and performs fixed point computations on loops.

The same rules that applied for the first phase also apply here, with one minor difference. Names on the left hand side of assignments are labeled as `AVAR` instead of `VAR`. When results are merged, labels for the same name are merged using the lattice depicted in Figure 11. Again incompatible labels can arise, both from these rules and from merging results, in which case the labels become $\top$.

5.3 Discussion

The goal of the Name Resolution Analysis is to estimate MATLAB's semantics for determining how a name is resolved and if it refers to a variable, if that variable has a value. In this section we discuss the effectiveness of this analysis by describing its accuracy.

There are two types of checks that we are concerned with. Checks to determine if a name is a variable or a function, and checks to determine if a variable is assigned at runtime. The former are particularly undesirable because they require the use of `eval`. The use of `eval` can make later analysis much less precise.

The MATLAB semantics for determining if a name is a variable or a function in functions are fairly static. Because of this, the Name Resolution Analysis is capable of accurately determining all names that are variables. This allows the compiler to eliminate all runtime checks for this property. By eliminating those checks the compiler also eliminates all uses of `eval`.

The analysis can also be fairly successful in eliminating runtime checks to determine if a variable is defined in a function. Once again this is due to the more static semantics of functions. The analysis can determine that roughly half of the variable uses in our example programs are guaranteed to be well defined. Script semantics are more dependent on runtime behaviour. Because of this the Name Resolution Analysis is less successful at determining how names are resolved in scripts.

6 Related Work

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

AspectJ [9] was one of the main languages that popularized aspect-oriented programming. AspectJ provides array pointcuts functionality, such that a type name pattern or subtype pattern can be followed by one or more sets of square brackets to make array type patterns. However, the pointcuts of AspectJ do not support array objects in full. When an element of an array object is set or referenced, the corresponding index values and the assigned value are not exposed to the advice. The availability of such information can be very helpful in multiple ways, such as the ability to bounds-check the array, optimization of array usage and profiling related to arrays. The original AspectJ does not support any loop pointcuts.

Researchers have experimented with array and loop pointcut extensions to AspectJ using abc, an extensible AspectJ compiler [3].

Harbulot extended the set pointcut to capture arrayset.[3] In his proposal, the pointcut designator `args()` exposes both the array index value and the object being assigned to an array element, and the pointcut designator `target()` exposes the array object being assigned. However, this extension bases its implementation on treating an array element set as a call to a `set(int index, Object newValue)` method, and thus works only for one-dimensional arrays.

As compared to Harbulot's extension, ArrayPT [5] works for multi-dimensional arrays. The core of the implementation is a finite-state machine based pointcut matcher that can handle arrays of multiple dimensions in a uniform way. They took the standard field set pointcut as the basis and developed this extension on the top of it. All array field set join points are treated as having a variable number of arguments: the sequence of index values and the value the field is being set to. At a join point, these values can be obtained using an `args()` pointcut designator and then passed to the advice for further processing. It enables the programmer to perform selective matching on any number of specified indices.

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

Another extension to the abc compiler, LoopsAJ [7], provides AspectJ with a loop pointcut. Loop selection is a major issue here, because unlike other pointcuts for variables and functions, loops don't have a named identification associated with them. In AspectMatlab, loop patterns are equipped with a facility to match the loops based on the variables being used in loop headers. Certain context exposure is provided to make the advice more effective.

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

AspectCobol [11] is inspired from AspectJ in many ways, but it incorporates original techniques for join point identification and context capture. AspectCobol's design strongly suggests that join point reflection on the join point shadow should be viewed as part of the pointcut as opposed to using reflection in the advice code. AspectCobol allows one to extract such details from the join point. The extraction is described as part of the pointcut designator, while the results are bound to variables for subsequent use in the advice code. Hence data is extracted from the shadow of the join point, i.e., the static program context that belongs to the join point.

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

There has been some effort made to introduce aspect-oriented features in MATLAB. Joao M. P. Cardoso, et al. [8] suggest various useful AOP features, especially those to specify different numeric data types. They have also pointed out the importance of AOP for MATLAB and their work suggests some further use cases. However, our approach includes both general-purpose aspects and specific patterns for scientific applications, as well as a complete and extensible language specification and open-source compiler, including analyses for the dynamic properties of MATLAB.

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

7 Conclusions and Future Work

In this paper we have introduced a new aspect-oriented scientific language, AspectMatlab. By providing clear extensions to an already popular language, MATLAB, we hope to make aspect-oriented programming accessible to a new group of programmers including scientists and engineers.

Our AspectMatlab language provides the basic patterns (pointcuts) that one expects, but also focuses on providing new patterns that work well for array accesses and loops. We have also designed and implemented the *amc* compiler for the new language. The *amc* compiler is designed to be easily extensible, so that other researchers can easily experiment with other new features useful for scientists. The compiler is a source-to-source compiler, producing ordinary MATLAB as its output. This means that any MATLAB system can be used to execute the woven code. The compiler is freely available at `www.sable.mcgill.ca/mclab/aspectmatlab`. Example programs and the generated code for them are also available.

AspectMatlab presents some challenges for producing correct and efficient woven code. We have described our approach to weaving, including our approach to `around` advice, and we provided static flow analyses that enabled us to reduce the number of dynamic checks required in the woven code. We plan to continue to measure the overheads and design further optimizations on the woven code.

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

In our future work we intend to continue to evaluate and improve the performance and functionality of AspectMatlab. We also intend to teach the language to scientists in different disciplines and to integrate new language features that would be useful to them. We will also continue to build our library of example use cases, which will be publicly-available.

References

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

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

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *AOSD*, pages 87–98. ACM Press, Mar. 2005.

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

[5] K. Chen and C.-H. Chien. Extending the field access pointcuts of AspectJ to arrays. *Journal of Software Engineering Studies*, 2(2):93–102, June 2007.

[6] T. Ekman and G. Hedin. The Jastadd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

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

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

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072, pages 327–353, 2001.

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

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

[12] J. D. F. Richard L. Burden. *Numerical Analysis, eigth edition.* Thomson Books/Cole, Belmont, California, 2005.

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