McGill University
School of Computer Science
Sable Research Group

The Importance of Being Extendable, A Crucial Extension for Aspect-Matlab

Sable Technical Report No. 2010-7

Olivier Savary B., McGill University
Prof. Laurie J. Hendren, McGill University

June 29, 2011

Contents

List of Figures

List of Tables

Abstract

AspectMatlab is an aspect-oriented extension of the MATLAB programming language. In this paper we present some important extensions to the original implementation of AspectMatlab. These extensions enhance the expressiveness of the aspect pattern definition and widen support to certain MATLAB native functions. Of particular importance are the new operator patterns, which permit matching on binary operators such as "*" and "+". Correct support for the MATLAB "clear" command and the "end" expression were also added. Finally, we documented previously hidden features, most notably a negation operator on pattern. This new extended version of MATLAB thus supports important added functionality.

1 Introduction

Aspect-Oriented programming is a paradigm based around a desire to separate the core of a program from its supporting features. These secondary features are encapsulated in functions in an aspect-source file, alongside with patterns defining the positions("before","around" or "after" nearly any expression) at which these functions should act in the source-code. Contrary to object-oriented programming, it demands little or no modification to the source-code, encapsulating its extension and other crosscutting concerns into another file. For that we felt it was a particularly relevant language extension to develop for MATLAB, given that scientific programmers often reuse MATLAB code by making small modifications. Another strong reason motivating this was its usefulness in debugging and profiling, when the programmer wants to get information about, for example, the program flow or the value of a variable throughout its execution.

From these concerns was born AspectMatlab, an aspect-oriented extension to the programming language MATLAB, developed as part of the McLab project. It was developed by T. Aslaam, as part of his Master Thesis at McGill, and was released as a compiler (Aspect-Matlab Compiler) in 2010 [1]. While this first release already had an extensive portion of the MATLAB grammar covered, and did so while introducing many solutions to Aspect-Oriented Programming in a dynamic language, certain functionalities required future work. Fortunately, extensibility was of primary concern in the development of the compiler and of the language, as tools and documentation were in place for the time when its vanilla coverage of the MATLAB language would not be sufficient for the more demanding user.

Following the release, it was brought to our attention that a certain number of extensions to the language itself could facilitate the adoption of the compiler, both by extending the coverage of the MATLAB language, and by providing useful patterns to our targeted user base, the scientific programmers.

These extensions can be divided into two categories, the first being rewriting of MATLAB Keywords, which were omitted in the first release for their lack of consistency when simplified or inlined. The keyword "end", used as an array index, and "clear", compilatric nightmare of dynamicity, are now dealt with dedicated analysis which rewrite portions of the code to accommodate for MATLAB semantic oddities. Detailed information about these keywords is provided in the section "Transformations". The second series of extension is ones made to the language itself, adding two pattern types to the ones already there, sharpening our ability to create patternmatching-specific language constructs, with the option to negate a pattern, making it match on every joint point unmatched by the pattern, and providing ways of matching directly on arithmetic operation by means of an "op" pattern. Their definition and usage are described in the "Patterns" section.

This report concludes with a short discussion about our views on languages and compiler extensions. Throughout the rest of this report, we will refer to T. Aslaam's Master Thesis "Aspect-Matlab", referenced in [1], as AMC.

1.1 Acknowledgment

Acknowledgment and thanks ought to be given to our research supervisor, Professor Laurie Hendren, for her generosity with her time and knowledge, and for giving us the opportunity to work in her research laboratory over the course of the summer. Another thanks is given to Toheed Aslaam, for his guidance in finding relevant extensions to work on, and for his inspirational work as the

main developer of the Aspect-Matlab Compiler. Finally, we would like to thanks Julie Langmann and Anton Dubrau for their advices on the writing and correction of this report.

2 Transformations

The following rewriting analysis were motivated, as indicated earlier in this report, by the absence, in the initial release of AMC, of support for certain keywords in MATLAB, either by lack of static information about the information they convey, as it is the case with the "end" keyword, or by their potential to break the weaved code, such as "clear", which can remove the elements introduced by AspectMatlab from the workspace. Implementing those new features facilitates the adoption of AspectMatlab by making it more respectful of MATLAB's syntax.

2.1 End

The keyword "end" is used in MATLAB both to terminate a block of code, such as in an if-then-else construct, and to refer to the last element of an array. In the latter, "end" is used as an index in an array access, and is equivalent to referring to the last element of the dimension it is written in. In the case where the keyword is embedded in multiple array access and function calls, "end" refers to the last element of the accessed row in the closest enclosing array. Assuming a workspace containing a function called foo and two arrays, A, of size 2x2(accessed in the example as a 4x1 via linear indexing) and B, of size 3x3, we can say that the following two are equivalent. The first end refers to the last row of B, that is, the 3rd row. The second end refers to the last element of A(accessed, as mentioned earlier, as a 4x1 array). The last end is the last element of the 3rd dimension of B, accessed as a 3x3x1 array.

```
B(end,A(foo(end)),end);
B(3,A(foo(4),1);
```

Two difficulties arise from the unparameterized nature of the call to "end". First, certain simplifications, in the initial release of Aspect-Matlab, take out the arguments from the call and replace it by a temporary variable. In such event, "end" is pulled out of its enclosing array and loses its meaning. Fortunately, there exists a parameterized version of "end", by use of the function "builtin". The first parameter required for "builtin" is the name of the desired function, "end" in our case. The rest of the arguments depend on the desired function, and the builtin version of end has 3 arguments: the array it is referring to, the position in the array access, and the order of the array as accessed in the call. When a MATLAB array is indexed with fewer indices than it has dimensions, the last index linearly accesses all remaining dimensions. Thus, in the previous example, the two dimensional matrix A can be accessed with one index only, giving access to all four elements.

Other alternatives, such as replacing "end" with the correct index, or writing a function evaluating "end" at runtime, that could be included in all AspectMatlab compiled files. The first was rejected after looking at the accessible information at compile time. Even if we know about an expression being an array, we might not be certain of its size, as variable are undeclared in MATLAB, and being morphed to accommodate the assignments to them. Moreover, the speed difference between "end", "builtin-end" and the indices was negligible, there was an significant overhead when we tried

to implement "end" as a function. Therefore, we decided to replace all calls to "end" as an array index by a call to the parameterized version of the same function.

Using the same workspace as in the previous example, the same code would now look like

```
B(builtin('end',B,1,3),A(foo(builtin('end',A,1,1)), builtin('end',3,3));
```

The second difficulty comes directly from this solution, where the array enclosing "end" must be determined at compile-time. Since MATLAB syntax for function call and array access are similar, it is sometimes undecidable, with our current analysis, to determine statically if a certain expression is an array or a function [2] . In the event of enclosed undecidable expressions between the call to "end" and the outer expressions, a runtime check is used to determine the type of these expressions inside an if-then-else construct. A temporary variable is initialized inside this construct and replaces the call to "end".

For instance, with a workspace containing an array B, of size 3x3, and two unknown items(variables or functions), foo and A, the initial

```
B(end,A(foo(end)),end);
```

would become

```
if isvariable(foo)
  AM_tempEnd0 = builtin('end',foo,1,1);
elseif isvariable(A)
  AM_tempEnd0 = builtin('end',A,1,1);
else
  AM_tempEnd0 = builtin('end',B,2,3);
end
B(builtin('end',B,1,3),A(foo(AM_tempEnd0)),builtin('end',B,3,3));
```

Figure 1 Example of "end" transformation

Such is the current state of our "end" rewrite. All rewritten "end" can be simplified while conserving all information related to its use but are kept in their initial position whenever possible, to favor readability.

2.2 clear

The keyword "clear" is used in MATLAB to remove items, such as object handles, loaded functions or instantiated variables, from the workspace [4]. It can be presented in the source-code with different arguments indicating what item or group of items should be cleared or alone, in which case it will remove all the items in the workspace. AspectMatlab adds a certain number of variables and objects to the workspace to keep track of certain dynamic proprieties of a program at runtime, and to be able to call the method contained in the aspect file. Because of the vulnerability of these variables to "clear", and of their importance in the correct execution of a program, misuses of "clear" could break the weaved program. To protect these AspectMatlab-specific runtime variables, we wrote an analysis that determines which items are susceptible of being removed from the workspace by the call to "clear" It then generates and weaves in codes that keeps these runtime variables alive while being semantically equivalent to the "clear" in the source-code.

This analysis takes place before weaving aspects into the parsed source-code. Once a "clear" statement is found we weave code that protects AspectMatlab global or local variables based on estimation which items are to be cleared. Difficulties arise from the possibility of using string literals as an argument, which can be created and filled at runtime. In these situations we protect all AspectMatlab items as if the call cleared all workspace items.

The protection itself is based on different levels of environment, where we promote and demote items to ensure that they are not to be affected by clear at this level. These items are then restored to their initial position, after "clear" has been applied. Calls that clear multiple levels and types of items are divided up. While reproducing the same effect, this allows us to weave-in the appropriate protections in-between calls. All the temporary items created by this analysis, identified with the usual "AM" prefix, are then thoroughly cleared, thereby conserving only the items from the source-code and the normal AspectMatlab ones. As an example, a call to "clear global" would be dealt with:

- Declaring temporary local variables.
- Assigning the AspectMatlab global variables to their newly-declared, local equivalent.
- Adding the statement "clear('global');"
- Restoring the AspectMatlab global variables to their previous state using the local copies.
- Clearing the temporary local variables

Figure 2 Stages in the transformation of "clear"

The resulting code then follows.

```
AM_GLOBAL_B = AM_GLOBAL;
clear('global');
global AM_GLOBAL;
AM_GLOBAL = AM_GLOBAL_B;
clear('AM_GLOBAL_B);
```

Figure 3 Example of "clear" transformation

Despite this not being the most elegant solution, as it adds many lines of code to the already bloated processed source file, it is simple, and requires no adaptation from the MATLAB user.

3 Patterns

3.1 Negative Matching

The AspectMatlab Compiler was initially released with different combinators such as logical AND("&") and OR("|") to compound patterns [1] , in addition to the primitive ones such as set and call. However, we felt that the compiler lacked the negation operator, as seen in other Aspect-Oriented Languages such as AspectJ [3].

Such an operator makes a pattern match with every join points that wouldn't have been picked by the initial pattern. It was determined that this interpretation is favorable to the one where the pattern matches every join points of the type of the negated pattern, but not matched by it, for it follow closely the logical definition of negation. We see its use in the definition of complex patterns, along with compound patterns. The grammar rule for logical negation, as written by T. Aslaam, is given bellow.

```
aspect ShadowMatch {
  [...]
 eq NotExpr.ShadowMatch(String target, String pattern, int args, ASTNode jp) {
    return !(getOperand().ShadowMatch(target, pattern, args, jp));
  }
  [...]
 }
```

Figure 4 Grammar rule for '∼'

As shown below, one could define a pattern matching on all calls to the function foo, except those with 2 arguments:

```
patterns
  callToFoo: call(foo) & ∼(foo(∗,∗));
end
```

Figure 5 Example of uses for the negated patterns .

Most of the code used to implement negative matching was already present, although undocumented, in the compiler. Our contribution to this task is limited to the activation, debugging and documentation of the feature. It should be noted that the symbol '∼' was used rather than the more common '!' for parsing reasons, '!' being already used in the header of aspect files and breaking the pattern construct. Moreover, '∼' is the MATLAB operator for logical negation, in line with the Principia Mathematica, so that our pattern feels in continuity with MATLAB syntax.

3.2 Operator pattern

T. Aslaam's AMC memorable chapter "Conclusions and Future Work" [1] mentions, as possible improvement to AspectMatlab, patterns cross-cutting on arithmetic operations. As seen in D.3 and D.4, It was necessary to rewrite all arithmetic operations to their equivalent function forms (for example "+" to "add"), because AspectMatlab had no pattern to match operators. This reduces the portability of the aspect code, and more generally the usability of the compiler, forcing the user to manually refactor his source-code. Keeping in mind the goals of "performance, usefulness and accessibility" [1], and seeing as many of the scientist use cases shown in it had to be rewritten to match on the function version of operators, it is clear that the implementation of this pattern was of foremost importance.

3.2.1 Modification and Addition to the Language Definition

We added a new primitive pattern, "op", which captures join points at arithmetic operations. The syntax closely follows the canon of AspectMatlab's release, with a single pattern selector, either a predefined keyword matching on multiple operators, or an operator itself, enclosed in parenthesis. As with all other primitive patterns, they can be combined or negated with the appropriate logical patterns. The following are different examples of operator patterns:

```
patterns
  plusOp : op(+);  %matching on addition
  timesOp : op(.*)||op(*) %matching on matrices or elementwise multiplication
  matrixOp : op(matrix); %matching on all linear-algebra operator
  allOp : op(all) & ~op(-); %matching on all arithmetic operator except minus
end
```

Figure 6 Example of uses for operator patterns

MATLAB's arithmetic operators can be separated in two types, the first being Matrix operations, "defined by the rules of linear algebra" [5] , and the second being Array operations, "carried out element by element" [5] . The keywords "matrix" and "array" can be used to declare a pattern matching on all operators of each category. Finally, "all" match on both Matrix and Array operation.

	function	enclosing type
+	plus	matrix
-	minus	matrix
*	mtimes	matrix
.*	ediv	array
/	mdiv	matrix
./	ediv	array
\	mldiv	matrix
.\	eldiv	array
^	mpow	matrix
.^	epow	array

Table I MATLAB Arithmetic Operators

As with the other patterns, we define a number of selectors capturing the context of joint= points, and reflect them in the actions. The selectors chosen in the action declaration contain information about the matched operation such as its position in the source-code, the name of its operands, &c. Below is a list of them, to be read as an addendum to the table "Context Selectors with respect to Join Points", in the introductory thesis behind AspectMatlab. It is similar to the ones defined for the "call" pattern, except for the omission of "obj" to get the function handle.

3.2.2 Simplification and weaving

Once properly matched, we must verify that its position in its parental statement exposes the weaving point at which we will later weave the action, and, if it is not the case, refactor the code to make it so. Each binaryExpr has three weaving points, and only appropriate ones are exposed by our transformations.

	op
args	operand(s)
obj	-
newVal	-
counter	number of arguments(unary or binary op.)
name	name of the entity matched
pat	name of the pattern matched
line	line number in the source code
loc	enclosing function/script name
file	enclosing file name
aobj	-
ainput	name of the operand(s)
aoutput	-
varargout	cell array variable used to return data from `around` action

Table II Context Selectors with respect to Join Points

It is important to note here a change in the flow of the compiler. Where AMC saw all expressions as "potential match, [would] there exists a pattern in the pattern list" [1] , and thus exposing all weaving-point in the case of complex expressions, we delay refactoring at a point after the matching, exposing only the relevant weaving-points. By limiting the number of transformations and the use of temporary variables, we maintained the code readability and the form of the arithmetic formula in the source code.

The implementation of these code refactorings is straightforward: BEFORE weaving-points are exposed by pulling up expressions under the binaryExpr, where with AFTER weaving-point we do so with the binaryExpr itself. AROUND are dealt with by simplifying both the binaryExpr and the expressions under it, so that the effect of replacing it by the action is limited to its own. [1]

The actions are then weaved at their appropriate position, next or onto the matched operator. In the case of multiple patterns matching at the same join point, the precedence order defined in the documentation of the initial release [1] is respected, with woven advice appearing in the order they were defined in the aspect source file.

3.2.3 Scientific Use Cases

Below is included an aspect inspired by the "unit" aspect , included in AMC as Figure 3.7. It is present here as a demonstration of simplifications applied on arithmetic operations while using the AMC operator pattern. In this MATLAB source code, we compute a Quetelet index(bmi) from a previously defined height in feet and inches, and a weight given in kg.

```
%variables alive: kg, feet, inches

height = (feet+inches/12)*0.3;

%bmi given in SI units
bmi = kg/(height)^2

disp(bmi);
```

Figure 7 Extract from bmi.m, MATLAB code

The following example shows a first application of an aspect, defined using the pattern "plusOp", on "bmi.m". The division "inches/12" has been simplified out of its initial position to expose the "before" pointcut of the addition in "height". The gain in readability of our just-in-time simplifications, compared to a naive join points exposure, is clearly visible here, where most of the arithmetic expression are kept intact.

```
patterns
    plusOp: op(+);
end
actions
    act : before plusOp
          ....
end
```

Figure 8 Example of an aspect 1, AspectMatlab code

```
%variables alive: kg, feet, inches

AM_temp1 = inches/12;
AM_Global.myAspect.myAspect_act([...]);
height = (feet + AM_temp1)*0.3;

%bmi given in SI units
bmi = kg/(height)^2;

disp(bmi);
```

Figure 9 Extract from bmi.m, Weaved Matlab code

Similarly to the previous example, this one shows the application of an action, defined using the pattern "allOpmDiv", on "bmi.m". This action is to be called after each operation, except division. It matches on the addition in "height", and on the exponentiation in "bmi", both of which needs to be simplified to expose the "after" pointcut.

```
patterns
    allOpmDiv: op(all) & (~op(/) & ~op(*));
end
actions
    act : after allOpmDiv
          ....
end
```

Figure 10 Example of an aspect 2, AspectMatlab code

```
%variables alive: kg, feet, inches
AM_temp1 = feet + inches/12;
\textbf{AM_Global.myAspect.myAspect_act([...]);}
height = AM_temp1*0.3

%bmi given in SI units
AM_temp2 = height^2
AM_Global.myAspect.myAspect_act([...]);
bmi = kg/AM_temp2;

disp(bmi);
```

Figure 11 Extract from bmi.m, Weaved Matlab code

4 Conclusions and Future Work

Following the completion of the noted T. Aslaam's Thesis, and the subsequent release of the AspectMatlab Compiler, our research project was conceived with the constant attention to the design choices behind AspectMatlab, so that our extensions do not denature the core of the compiler while acting on their own, limited, whereabouts.

The code for each extension is comprehensively identified and commented in the compiler source-code. This notion is perfectly illustrated by our decision to position the transformation on "end" and "clear" as an analysis, directly in the method taking care of the compiler flow, rather than inlined in the classes of relevant expressions. Doing so, we avoid weakening the core of the compiler with our optional features, leaving its performance and effects intact. A programmer's code often have a distinct, and unique style. Confusion regarding the effect of a code arises not only for bad styling and commenting, but also from a poor cohesion due to multiple styles being present side-by-side. One should use, as much as possible, the tools given by the previous programmer to interface with his new feature, rather than integrating it in the initial source-code. This cleanness will benefit future work on the compiler.

Nevertheless, our views on the user-visible part is diametrically opposed, with a desire for our rewriting to produce code of congruous nature with the previously generated code, for a seamless AspectMatlab code in the weaved source file. This leaves the end-user with a much more practical AspectMatlab/ MATLAB distinctive division, in continuity with the previous release. The same attention motivated the definition of our extensions on pattern, for example by limiting our patterns on operators to one operator or keyword, leaving the more complex pattern declarations to the appropriate compounding, in a similar fashion to the previously defined pattern types such as "call".

This bring us to the main point of this discussion, and of this report, for the knowledge and experience brought by this research opportunity offset, by far, our humble contributions. Compilers are extremely complex programs, often containing thousands of lines of code, and relating to many fields of computer science. More so, they support an image of a programming language, which itself evolves and expands throughout new research and desired uses. For that it is essential to develop tools and mechanics among them in such way where extending them is simple. It can then follow, whenever it is possible, the evolution of the language it supports, thus extending its

lifetime, allowing researchers to concentrate on cutting-edge research in precise areas rather than on the programming itself.

And for that compiler researchers prefer the more redeeming areas of compiler developments, be it optimization, analysis, &c, to the contrary of the seemingly repetitiveness and triviality of programming the core itself, I've now realized for the first time in my life the vital Importance of Being Extendable.

References

[1] Toheed Aslaam. Aspectmatlab: An aspect-oriented scientific -programming language. Master's thesis, McGill University, February 2010.

[2] Doherty J. Dubrau A. Aslam, T. and L. Hendren. Aspectmatlab: an aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, New York, NY, USA, 2010. ACM.

[3] AspectJ . Pointcuts. http://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html, 2003.

[4] MathWorks. Clear. http://www.mathworks.com/access/helpdesk/help/techdoc/ref -/clear.html, 2010.

[5] MathWorks. Matrix and array arithmetic. http://www.mathworks.com/access/helpdesk -/help/techdoc/ref/arithmeticoperators.html, 2010.