Typing Aspects for MATLAB*

Laurie Hendren
Sable Research Group, School of Computer Science, McGill University
hendren@cs.mcgill.ca

ABSTRACT

The MATLAB programming language is heavily used in many scientific and engineering domains. Part of the appeal of the language is that one can quickly prototype numerical algorithms without requiring any static type declarations. However, this lack of type information is detrimental to both the programmer in terms of software reliability and understanding, and to the compiler in terms of generating efficient code.

This paper introduces the idea of adding typing aspects to MATLAB programs. A typing aspect can be used to: (1) capture the runtime types of variables, and (2) to check runtime types against either a declared type or against a previously captured runtime type. Typing aspects can be can be used: (1) solely as documentation, (2) to log type errors, or (3) to catch type errors at runtime.

Categories and Subject Descriptors

D.3.3 [**Programming Lang.**]: [Language Constructs and Features]

General Terms

Experimentation, Languages, Performance

Keywords

Typing aspects, Dynamic type assertions, MATLAB

1. INTRODUCTION

MATLAB is a popular dynamic programming language used for scientific and numerical programming with a very large and increasing user base. The most recent data from MathWorks shows that the number of users of MATLAB was 1 million in 2004, with the number of users doubling every 1.5 to 2 years.[1] Certainly it is one of the key languages used in education, research and development for scientific and engineering applications. There are currently over 1200 books based on MATLAB and its companion software, Simulink (`http://www.mathworks.com/support/books`). This large and diverse collection of books illustrates the many scientific areas which rely on computational approaches and use MATLAB.

One of the key features of MATLAB is that it has no statically-declared types. The lack of type declarations is often considered an advantage for fast prototyping. However, having no statically-declared types also has many disadvantages, including negative impacts on developing reliable and reusable programs, and negative impacts on performance.

From the programmer's point of view, a MATLAB function often actually has many implicit assumptions about the types of variables, especially parameters. For example, it may be assumed that a parameter "`n`" is a scalar, a parameter "`a`" is a two-dimensional matrix, or that a parameter "`f`" is a function handle (a reference to a function and closure). If the function is called with arguments of the wrong types, runtime errors or unexpected results may occur. Thus, from both the reliability perspective, as well as program reusability, making these assumptions explicit and checkable would be beneficial.

The lack of static types also negatively impacts performance. Although the original MATLAB systems were interpreted, both the proprietary Mathworks system[2] and the open-source McLab[1] systems now contain JIT compilers. JIT compilers, and ahead-of-time compilers, require type information to produce efficient code.

In this paper we introduce the idea of MATLAB typing aspects. The idea is that one can annotate MATLAB functions with aspect type statements (henceforth referred to as **atype** statements) which serve two main purposes. First, an **atype** statement is used to verify that the runtime type matches a specified type, where the specified type may be given at various levels of refinement. Secondly, an **atype** statement is used to capture all or part of the runtime type, which can be used in subsequent **atype** statements.

Since **atype** statements are not part of standard MATLAB, but are rather a declarative way of specifying dynamic type checks, a weaver is required to convert the **atype** statements to native MATLAB. The weaver converts each **atype** statement into standard MATLAB code which performs the appropriate check and action.

We have identified three levels of woven target code, corresponding to different levels of runtime checking. The most rigorous level introduces dynamic checks which raise runtime errors when a type mismatch is detected. The middle level makes the dynamic checks, but only logs runtime type errors. The least rigorous level introduces comments, similar in spirit to the style of comments the MATLAB programmers often insert by hand.

The remainder of this paper is structured as follows. Section 2 gives an overview of the different types in MATLAB and provides some motivating examples. Section 3 introduces the syntax and

*This work supported by NSERC and the Leverhulme Trust.

[1]From `www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DSAL'11, March 22, 2011, Pernambuco, Brazil.
Copyright 2011 ACM 978-1-4503-0648-5/11/03 ...$10.00.

semantics of the proposed **atype** statements. Section 4 discusses related work, and Section 5 gives conclusions and future work.

2. BACKGROUND

The MATLAB programming language was originally designed as an interactive interface to numerical libraries and had only one data type, **matrix**.[2] Over the years the language has been extended, and it now includes a variety of data types, and is used for substantial programming projects. As illustrated in Figure 1, a MATLAB value can either be **data** or a function handle (**fnhandle**). Data comes in three varieties: **array**, **cellarray** and **struct**.

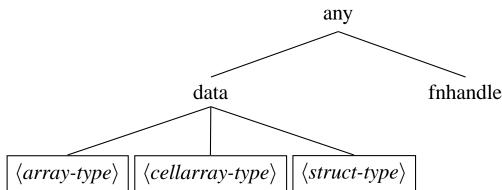

Figure 1: High-level MATLAB types

2.1 Arrays

Arrays are homogeneous (i.e. all elements have the same type) and the elements must be some scalar type (*double*, *int32*, *char*, ...). In particular, the elements of arrays cannot be handles, structs or cell arrays. Arrays have a shape and contents, and arrays are mutable. By default, the type of array elements is double, and even scalar variables are arrays (1×1 arrays). For example, a statement of the form "`a = 3`" defines a 1×1 array of type double, even though an integer literal was used in the statement. However, it is possible to explicitly create other types. For example, the statement "`b = uint64(3)`" creates a 1×1 array of unsigned 64-bit integers.

The base types in MATLAB can be ordered as presented in Figure 2.[3] The leaves in this ordering correspond to possible runtime types, whereas the interior nodes represent groups of related types, which we will use in the typing aspects. At the top (left-hand-side of the figure) of the ordering we use "`*`" to indicate all possible base types. As indicated by the second level of the ordering, MATLAB distinguishes between *numeric* types versus *char* and *logical* types. Strings are effectively $1 \times n$ arrays of *char*. For example, the statement "`s = 'cat'`" assigns a 1×3 array of characters to variable "`s`". Logical types are used to represent boolean values. For example, the statement "`b = isequal(3,2+1)`" would assign a 1×1 array of logical to variable "`b`".

Within the *numeric* types, there are two groups *real* and *complex*. Note that these types do not imply floating point numbers, but serve to distinguish values that are representing real numbers (one part) versus values representing imaginary numbers (two parts, the real and imaginary parts). Thus, it is perfectly possible to create a complex number in which the two parts are represented as unsigned 64-bit integers. For example, the statement ("`c = complex( uint64(3), uint64(4))`" creates a 1×1 array with type uint64:complex.[4] A real number can be represented either as an *int* or *float*, where *int* can be either *signed* or *unsigned*, and *float* can be either *single* or *double* precision.

2.2 Structs and Cell Arrays

Structs and cell arrays are heterogeneous and provide a way of aggregating data. Structs do not have explicit types but are constructed using calls like "`a=struct('x',exp1,'y',exp2)`", which would create a structure with two fields, "`x`" initialized to the value denoted by "`exp1`" and "`y`" initialized to the value denoted by "`exp2`". Each field can contain any type (array, handle, struct or cell array). Cell arrays have the same rectangular structure as arrays, but their elements are cells instead of numeric values, where each cell can contain any type. Thus cell arrays allow one to create heterogeneous and nested arrays. Cell arrays are accessed using "`a(...)`" which denotes the cells or "`a{...}`" which denotes the contents of the cells.

2.3 Function Handles

A function handle refers to a closure, where the closure consists of a reference to the function and a reference to a workspace that maps free variables to values. A function handle is created by either taking the handle of a named function (for example, "`h = @sin;`") or by creating a handle to an anonymous function (for example, "`h = @(x)(x+1);`").

2.4 Implicit types in MATLAB functions

Although some MATLAB functions can operate on values of any type (for example, the built-in function "`size`"), most functions actually have some implicit requirements about the types of their input and output parameters. If the function is provided with the "wrong" input types, it may produce undesired results or may fail in a completely non-obvious way.

Consider the function "`Ex1`" given in Listing 1. This function has one input parameter, "`n`", and one output parameter "`r`". The comments describe the purpose of the function and from these comments one could deduce that "`n`" should be some sort of scalar number, and that the output will be a vector.

```
1 function [ r ] = Ex1( n )
2 % Ex1(n) creates a vector of n values containing
3 %   the values [sin(1), sin(2), ..., sin(n)]
4   for i=1:n
5     r(i) = sin(i);
6   end
7 end
```

Listing 1: Example MATLAB function

Figure 3 shows an interactive MATLAB session, testing the behaviour of various input types for "`n`". The lines starting with "`»`" are user input. The lines starting with "`ans = `" are the results from evaluating the input expression.

Note that the first two examples in Figure 3 demonstrate the expected input and behaviour. Remember that values are by default double in MATLAB. The next five examples show the behaviour when other input types are used (int32, char, fnhandle, complex and logical). These all result in either a warning or an eventual runtime error. Not all of the error messages are simple to interpret, for example, the error message *"??? Undefined function or method '_colonobj' for input arguments of type 'function_handle"* is not easy to interpret. The last example shows how MATLAB often uses only part of an input. In this case the programmer has provided an

[2]See http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html.

[3]This ordering was created by examining user-level MATLAB documentation, in particular the set of pre-defined functions for testing types such as "`isinteger`".

[4]This type is represented in MATLAB as `class=double, attribute=complex`.

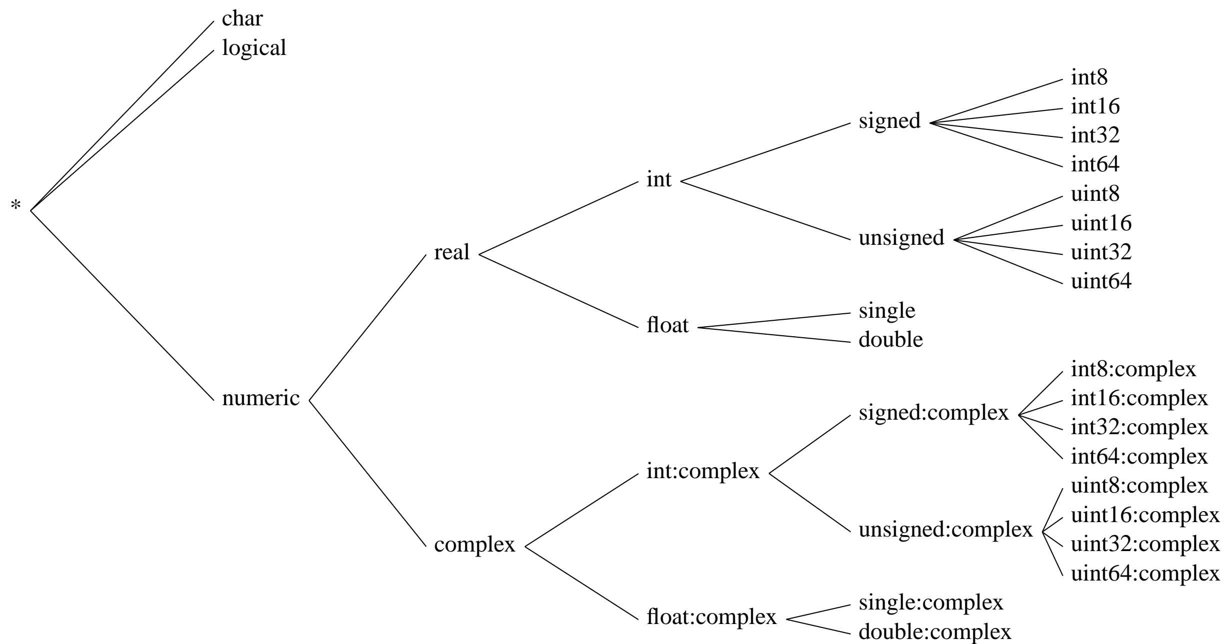

Figure 2: Ordering of base-types

```
>> Ex1(3)
ans = 0.8415    0.9093    0.1411

>> Ex1(2.3)
ans = 0.8415    0.9093

>> Ex1(int32(3))
??? Undefined function or method 'sin' for input
    arguments of type 'int32'.
Error in ==> Ex1 at 6
  b(i) = sin(i);

>> Ex1('c')
??? For colon operator with char operands, first
    and last operands must be char.
Error in ==> Ex1 at 2
  for i=1:n

>> Ex1(@sin)
??? Undefined function or method '_colonobj' for
    input arguments of type 'function_handle'.
Error in ==> Ex1 at 2
  for i=1:n

>> Ex1(complex(1,2))
Warning: Colon operands must be real scalars.
> In Ex1 at 2
ans = 0.8415

>> Ex1(true)
Warning: Colon operands should not be logical.
> In Ex1 at 2
ans = 0.8415

>> Ex1([3,4,5])
ans = 0.8415    0.9093    0.1411
```

Figure 3: Interactive session testing inputs to "`Ex1`"

array, "`[3,4,5]`" instead of the expected scalar. When this value is used by the colon operator, it uses only the first element of the array ignores the other values.

```
1 function y = sturm(X,BC,F,G,R)
2 % STURM Solve the Sturm-Liouville equation:
3 %   d( F*dY/dX )/dX - G*Y = R using linear  finite  elements.
4 % INPUT:
5 %   X  - a one-dimensional grid-point array of length N.
6 %   BC - is a 2 by 3 matrix [A1, B1, C1 ; An, Bn, Cn]
7  ...
8 % Alex Pletzer: pletzer@pppl.gov (Aug. 97/July 99).
9  ...
```

Listing 2: Example of types in comments

The "`Ex1`" example demonstrates that even though there may be a very simple type specification intended by the programmer, there is no simple way for the programmer to specify and check those types. Furthermore, the errors and warnings raised at runtime may not be simple to understand and may appear at a different program point than expected.

MATLAB programmers often provide comments which indicate the expected type of the input and output parameters. For example, consider the "`sturm`" function taken from a set of publicly-available applications, as shown in Listing 2.[5] Clearly the programmer has quite precise specifications for each input parameter, but has no formal way of specifying these requirements.

3. TYPING ASPECTS

Typing aspects are our solution to specifying and capturing relevant typing information in MATLAB programs. The key to our approach is the **atype** statement which is used both to capture the runtime types of variables, as well as perform a type check against a type specification. Each **atype** statement is composed of the name (or pattern) of a variable, and a type specification for that variable. The expected type can be very generic, or more specific, following the structure of MATLAB types as specified in the previous section.

[5]Source from `www.mathtools.net/sturm.m`.

Let us start with an example, adding typing information to the small example from Listing 1. Listing 3 gives a modified program, with the **atype** statements added.[6]

```
1 function [ r ] = Ex1( n )
2 % Ex1(n) creates a vector of n values containing
3 %   the values [sin(1), sin(2), ..., sin(n)]
4   atype('n','scalar of float');
5   for i=1:n
6     r(i) = sin(i);
7   end
8   atype('r','array [n.value] of n.basetype');
9 end
```

Listing 3: Example MATLAB **function with atype statements**

In general, we suggest one **atype** statement for each input and return parameter. The **atype** statements for input parameters should be inserted at the earliest point at which the parameter is definitely defined (in the case of an optional parameter, this may not be at the top of the function), and the **atype** statements for return parameters should be inserted at the last point before control returns from the function.

In Listing 3 the **atype** statement on line 4 specifies that "n" should be a scalar (syntactic sugar for **array** [1,1] **of float**) and that it should have a base type **float**. One could also use the more restrictive base type of **double**, but in fact the function is well-defined for both **double** and **single**. Note that as well as specifying the runtime type check, the **atype** statement also captures relevant context information about the value and types of the variable. In this example, we use the captured information "n.basetype" and "n.value" on line 8. Specifically, the **atype** statement on line 8 specifies the type of the return parameter, "r". This type specification says that "r" is a vector with size "n.value" and with base type equal to the runtime base type of the input parameter "n". Thus, a call of the form "Ex1(2.0)" will return a 1×2 array of **double** values, and a call of the form "Ex1(single(2.0))" will return a 1×2 array of **single** values.[7]

3.1 Syntax of atype statements

Figure 4 gives the syntax rules for the **atype** statement. At the top-level an atype statement specifies a ⟨*varname*⟩ and a type specification, ⟨*type-spec*⟩. A ⟨*varname*⟩ can be an identifier, or a pattern denoting a set of identifiers. The pattern can be the wildcard "*", or a string starting or ending with "*". This allows one to match on all variables, or all variables with specific prefixes or suffixes. The ⟨*type-spec*⟩ can be "any", which indicates any type, or ⟨*type-spec*⟩ can be something more specific. If the parameter should correspond to a function handle, then "fnhandle" should be used. To specify any kind of data (i.e. not a function handle), "data" can be used.

Quite often the programmer will want to specify the data type with more refinement than just "data". An ⟨*array-type*⟩ specification is either a "scalar" or "array", and an optional ⟨*base-type*⟩. The specifications for ⟨*base-type*⟩ follow the natural grouping of MATLAB types as specified earlier in Figure 2. Note that the programmer can choose the appropriate level of refinement. A base type of "*" is most general representing any base type, whereas "uint32" is a completely refined type representing unsigned 32-bit integers. Programmers may also want an intermediate refinement, for example "numeric" specifies any real or complex type, and "int" represents any integer type.

A "scalar" has implicit dimensions of 1×1, whereas an array may specify the dimensions. A dimension specification may be very refined, for example "[10, 20]" specifies a 10×20 array; or very general, for example "[...]" specifies any number of dimensions with any size.[8] One may also want intermediate refinement, for example "[*,*]" specifies a 2-dimensional array of any

⟨*atype-stmt*⟩ ::= atype '(' ''' ⟨*varname*⟩ ''' ',' ''' ⟨*type-spec*⟩ ''' ')'

⟨*varname*⟩ :== ⟨*identifier*⟩
| ⟨*identifier-pattern*⟩

⟨*identifier-pattern*⟩ :== '*'
| '*' '[a-zA-Z0-9]+'
| '[a-zA-Z][a-zA-Z0-9]*' '*'

⟨*type-spec*⟩ ::= any
| ⟨*data*⟩
| fnhandle
| ⟨*identifier*⟩ . type

⟨*data*⟩ ::= data
| ⟨*array-type*⟩
| ⟨*cellarray-type*⟩
| ⟨*struct-type*⟩

⟨*array-type*⟩ ::= scalar [of ⟨*base-type*⟩]
| array [⟨*dims*⟩] [of ⟨*base-type*⟩]

⟨*base-type*⟩ ::= '*'
| char
| logical
| ⟨*numeric*⟩
| ⟨*identifier*⟩ '.' basetype

⟨*dims*⟩ ::= '[' ⟨*dim-list*⟩ ']'
| '[' '...' ']'
| '[' '...' ',' ⟨*dim-list*⟩ ']'
| '[' ⟨*dim-list*⟩ ',' '...' ']'
| '[' ⟨*dim-list*⟩ ',' '...' ',' ⟨*dim-list*⟩ ']'
| '[' ⟨*identifier*⟩ '.' dims ']'

⟨*dim-list*⟩ ::= ⟨*dim*⟩
| ⟨*dim*⟩ ',' ⟨*dim-list*⟩

⟨*dim*⟩ ::= '*'
| ⟨*integer-literal*⟩
| '<' ⟨*identifier*⟩ '>'
| ⟨*identifier*⟩ '.' value
| ⟨*identifier*⟩ '.' ⟨*identifier*⟩

⟨*cellarray-type*⟩ ::= cellarray [⟨*dims*⟩] [of ⟨*type-spec*⟩]

⟨*struct-type*⟩ ::= struct [with '{' ⟨*struct-field-list*⟩ '}']

⟨*struct-field-list*⟩ ::= ⟨*struct-field-type*⟩
| ⟨*struct-field-type*⟩ ',' ⟨*struct-field-list*⟩

⟨*struct-field-type*⟩ ::= ⟨*identifier*⟩ ':' ⟨*type-spec*⟩

Figure 4: atype syntax rules

[6] Note that we have required that each argument is given as a string as indicated by the quote marks. We have done this so that the **atype** statement can be implemented either through a weaver, such as the one in AspectMatlab [4], or can be implemented as a stand-alone library.

[7] In MATLAB the default type of literals is **double** and vectors are represented by $1 \times n$ arrays.

[8] This is equivalent to giving no dimension specification.

size; "[10,*]" specifies a 2-dimensional array with exactly 10 rows and any number of columns; and "[10,20,...]" specifies an array with at least 3 dimensions, where the first two dimensions are "10" and "20". In addition, dimensions can use the notation "<n>" to capture the value of a dimension. Those values can then be used in subsequent **atype** statements. For example, consider the the specification in Listing 4 that expects an $n \times m$ input array "a", an $m \times p$ input array "b" and returns an $n \times p$ output array "r".

```
1 function [ r ] = foo( a, b )
2   atype('a','array[<n>,<m>] of real');
3   atype('b','array[a.m,<p>] of a.basetype');
4   % ...
5   % body of foo
6   % ...
7   atype('r','array[a.m,b.p] of a.basetype');
8 end
```

Listing 4: Capturing dimension values <n>, <m> and <p>

For each variable "x" matching an **atype** statement, the values "x.type" and "x.value" are captured, representing the runtime type and value of "x", as observed at the point of the **atype** statement. These values can be used in subsequent **atype** statements. For example, if one wanted to ensure that return parameter "r" had the same type as input parameter "a" in function "foo", one could use the specification in Listing 5.

```
1 function [ r ] = foo( a )
2   atype('a','any');
3   % ...
4   % body of foo
5   % ...
6   atype('r','a.type');
7 end
```

Listing 5: Using the captured type, a.type

In addition, for each **atype** statement matching array types, the dimensions and base type are captured. These can be used in many ways, for example to specify that the input parameter is an array of any integer type and the the output array is an array of the same dimensions one could use the specification in Listing 6.

```
1 function [ r ] = foo( a )
2   atype('a', 'array [...] of int');
3   % ...
4   % body of foo
5   % ...
6   atype('r','array[a.dims] of int');
7 end
```

Listing 6: Using the captured dimensions, a.dims

Returning to the example in Listing 4 we can see a use of the captured base type of "a". The base type of "a" is defined as **real**, which means that runtime base type can be any type below **real** in the ordering in Figure 2, for example **int32** or **double**. The **atype** statements for "b" and "r" specify the base types in terms of the runtime base type of "a". Thus, if "a" has runtime base type **int32**, then "b" and "r" should also have runtime base type **int32**.

Although most parameters in MATLAB programs are either arrays or function handles, some parameters are cell arrays or structs. In the case of cell arrays, the ⟨*base-type*⟩ part of the specification applies to all elements of the cell array. Since a cell array can be heterogeneous, the runtime check to verify these types can be expensive, and so should be used sparingly.

Listing 7 gives an example with two input parameters and one return parameter. The first input parameter, "ca", is an $n \times m$ cell array where each cell contains a struct with two **double** fields, "x" and "y". The second input parameter, "f", is a function handle. The return parameter, "r", is a $m \times n$ cell array with the same base type as "ca".

```
1 function [ r ] = foo( ca, f )
2   atype('ca', 'cellarray [<n>,<m>] of
3       struct with { x: double, y: double }');
4   atype('f',fnhandle);
5   % ...
6   % body of foo
7   % ...
8   atype('r','cellarray[ca.m,ca.n] of ca.basetype');
9 end
```

Listing 7: Function handle and cell array example

3.2 Semantics of atype statements

We have defined three levels of semantics for the **atype** statements. The least rigorous is to use them for documentation purposes. They provide a concise notation for documenting the intended types of parameters. However, our real intended purpose is to use them to capture runtime types and to check the type against the specification. The two more rigorous semantics require the three steps as described below.

Capture runtime type: If no variable matches ⟨*varname*⟩, then an unmatched variable error is raised. Otherwise, the runtime type *v.type* and value *v.value* of each variable *v* matching ⟨*varname*⟩ is stored in a type workspace, and subsequent **atype** statements can refer to these values. Furthermore, based on the stored type, subsequent **atype** statements can also query the dimensions and base type using *v.dims* and *v.basetype*.

Check against static specification: If the first step succeeded, then the runtime type *r* is checked against the static specification given in ⟨*type-spec*⟩. If the runtime type is not consistent with the specification, then a runtime type error is raised.

Bind free variables: If the runtime type is consistent, then any free type variables in the dimensions specification are bound to the actual values in the runtime type. These type variables exist in a namespace distinct from the ordinary program variables and may only be accessed in other **atype** specifications. Repeated free variables must have the same runtime values, so a specification of the form: "atype('a','array[<n>, <n>] of numeric'" defines a square matrix.

The difference between the two checking semantics is how errors are handled. In the less rigorous approach the errors are logged, but execution continues normally. This is useful if you merely want to observe potential type problems, but not change the functioning of the program. For the most rigorous approach the runtime type errors raise real runtime exceptions that will cause the program to terminate unless they are explicitly caught and handled.

3.3 Implementation strategies

There are two main implementation strategies. The one we favour is weaving via the AspectMatlab compiler. With this strategy, each

atype statement is first checked for syntactic and semantic validity. The syntax is as given in Figure 4. The semantic checks ensure that there is at least one matching variable for each **atype** statement and that any use of a type variable is defined along all paths to the use. The compiler then replaces the **atype** statement with inlined pure MATLAB code for each matching variable. For each variable *v*, the compiler captures the value *v.value* and runtime type *v.type* and performs the dynamic type check. If the type check succeeds, then it creates bindings for any free type variables. This inserted code is quite straight-forward and can use many MATLAB built-in primitives. The advantages of this approach include: (1) the fact that the final code is pure MATLAB, which can be run and shipped without any additional libraries, and it should have minimal impact on the JIT compiler; and (2) the specifications can be syntactically and semantically checked by the compiler.

The second approach is to implement **atype** as a MATLAB function. This is quite possible to do as MATLAB supports some rather invasive functions such as `eval`, `evalin` and `assignin`, which allow one to execute arbitrary MATLAB code and to modify the workspace of the calling function. The advantage of this approach is that a separate compiler, including the matcher and weaver, is not needed. The disadvantages are: (1) the overheads involved in extra function calls and the negative impact on the JIT due to using `evalin` and `assignin`; and (2) there are no syntactic and semantic checks performed by the compiler, checks must be done at runtime.

4. RELATED WORK

There are three main bodies of work related to this approach. This first is work done on static type and shape analyses for MATLAB such as the approaches used in McVM[5], MaJIC[3] and FALCON [10]. It is our hope that the **atype** statements could be used to both simplify and improve those static approaches. For example, by adding **atype** statements to key library routines, the type estimation for user programs using those libraries can be improved.

The second main body of work is type inference for other dynamic languages, which is often combined with some annotation framework to aid in the inference. Two excellent examples of this approach are the work by Furr et al. on static type inference for Ruby[8] and the work by Papi et al. on pluggable types for Java[9]. The key novelty of our approach is that **atype** statements have been designed to solve the kinds of array-based type problems that are specific to MATLAB and are implemented using a weaving-based approach.

The third body of work is work related to aspects. One might wonder if **atype** statements are truly aspects since the **atype** statements are explicitly inserted into the base program code and not defined separately as with traditional aspect declarations.[9] However, using the notion of aspects supporting quantification, as defined by Filman and Friedman [7], we can say that **atype** statements allow quantification over variables in a function. Another aspect-like approach used for supporting a pluggable type system was the use of sub-method reflection to implement a pluggable type system for Squeak[6].

5. CONCLUSIONS AND FUTURE WORK

This paper has proposed typing aspects which provide a mechanism for specifying, capturing and checking the dynamic types in MATLAB. The approach has been specifically tailored to the types and typical uses of MATLAB programs. The core of the technique is the **atype** statement which can be used to capture and check types at runtime. We have defined the syntax and semantics of the **atype** statement and presented two alternative approaches for the implementation. The implementation we favour is based on the AspectMatlab compiler, and we are currently working on the weaving-based approach using this implementation.

Our future work will focus on designing the minimal optimized instrumentation needed by the weaver. For example, only the values and types needed by subsequent **atype** statements need to be stored. We also hope to eliminate some dynamic checks based on static type analyses. Finally, we intend to use **atype** statements to aid in the documentation of standard MATLAB libraries, which will in turn help in type analysis of MATLAB programs as required by JIT compilers such as McVM and static MATLAB-to-Fortran compilers such as McFOR[1].

[9]Note that there is no technical reason why the **atype** specifications couldn't be given separately and woven "after the first definition of a specified variable" or "before returning from a specified function". However, we believe that putting the **atype** statement inline is the best form of documentation and is very natural for MATLAB programmers.

6. REFERENCES

[1] McLab: An Extensible Compiler Framework for MATLAB. Home page `http://www.sable.mcgill.ca/mclab/`.

[2] Accelerating MATLAB, 2002. `http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf`.

[3] G. Almasi and D. A. Padua. MaJIC: A MATLAB Just-In-Time Compiler. In *Languages and Compilers for Parallel Computing*. Springer Berlin / Heidelberg, 2001.

[4] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. AspectMatlab: An Aspect-Oriented Scientific Programming Language. In *Proceedings of 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, March 2010.

[5] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *International Conference on Compiler Construction*, pages 46–65, March 2010.

[6] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):275–295, 2007.

[7] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical Report Technical Report 01.12, RICAS, 2001. Presented at the Workshop on Advanced Separation of Concerns, OOPSLA 2000.

[8] M. Furr, J. hoon (David) An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[9] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[10] L. D. Rose, K. Gallivan, E. Gallopoulos, B. A. Marsolf, and D. A. Padua. FALCON: A MATLAB Interactive Restructuring Compiler. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 269–288, London, UK, 1996. Springer-Verlag.