To appear in the **Proceedings of the 10th ACM International Conference on Supercomputing - ICS'96**. Philadelphia, PA, May 1996

# A MATLAB to Fortran 90 Translator and its Effectiveness

Luiz De Rose and David Padua

March 1996
CSRD Report No. 1462

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, Illinois 61801

# A MATLAB to Fortran 90 Translator and its Effectiveness *

Luiz De Rose and David Padua
Center for Supercomputing Research and Development
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, U.S.A.
({derose,padua}@csrd.uiuc.edu)

## Abstract

In this paper, we describe the inference mechanism used by the FALCON system to translate MATLAB[1] programs to Fortran 90. FALCON is a programming environment for the development of scientific libraries and applications. The objective of the MATLAB compiler is to allow program development to take place in a user-friendly, interactive environment without sacrificing performance. FALCON's inference mechanism combines static and dynamic inference methods for intrinsic type, rank, and shape inference, and is supported by a sophisticated symbolic value propagation algorithm. Experimental results show that FALCON's MATLAB compiler can generate code that is over 1000 times faster than MATLAB on a uniprocessor SGI Power Challenge, and is often as fast as hand-written Fortran programs.

## 1 Introduction

The development of software for scientific computation on high performance computers is a difficult and time-consuming task. We are studying strategies to facilitate this process, and are implementing them in FALCON [8, 9], a programming environment for the development of scientific libraries and applications.

Program development in FALCON starts with a prototype and proceeds with a sequence of automatic and interactive transformations until an effective program or routine is obtained. The prototype and intermediate versions of the code are represented in the MATLAB language [15]; we chose MATLAB over other interactive array languages, such as APL [12], because of its popularity and its clean, well-structured design.

MATLAB does not require variable declarations which, in the opinion of some, simplifies the programmer's task. However, the downside is that MATLAB, like most languages without declarations, is interpreted, and therefore its performance is often much slower than the that of compiled languages. To circumvent this problem, FALCON includes a MATLAB to Fortran 90 translator. This paper describes this translator and presents some data on its effectiveness. Our ultimate goal is to generate parallel code by integrating FALCON with Polaris [4], a parallelizing compiler developed at Illinois. In this paper, we focus exclusively on the techniques we use to generate effective sequential code. These techniques are either *static* inference mechanisms used to generate declarations at compile time, or *dynamic* strategies that are applied at execution time when a lack of statically available information prevents the automatic generation of a particular variable's declaration.

Type inference algorithms, including some developed for languages that are quite similar to MATLAB such as SETL [19] and APL [5, 6], have been developed in the past. In the work reported here, we apply these type inference techniques with extensions from approaches that were originally developed to analyze and represent array accesses in Fortran [20]. These extensions are useful in the case of MATLAB, where arrays are often built using Fortran-like loops and where different sections of an array may be built in different program regions. In contrast, the techniques developed for APL assume that arrays are usually built by a single array operation.

The rest of this paper is organized as follows: FALCON's static inference mechanism is presented in Section 2; the dynamic strategy is discussed in Section 3; some experimental results are presented in Section 4; related work is discussed in Section 5; and, finally, our conclusions are presented in Section 6.

## 2 The Static Inference Mechanism

To generate Fortran 90 declarations, FALCON analyzes an Abstract Syntax Tree (AST) representation of a MATLAB program in an attempt to identify for each variable the following three properties: intrinsic type (i.e. INTEGER, REAL, COMPLEX, or LOGICAL); rank (i.e. SCALAR, VECTOR, or MATRIX); and shape (i.e. size of each dimension). These variable properties are estimated using a forward/backward propagation strategy [1].

Our inference mechanism extracts the initial type information from four sources: program constants whose intrinsic type, rank, and shape are statically known; input files; operators; and functions. If the program loads a variable, FALCON can extract from a sample external file the initial intrinsic type and rank[2] of the variable[3]. Variable shapes are not extracted from the input files because they are much more likely than intrinsic type and rank to differ between runs. However, the translator propagates shape information in terms of input values. Built-in MATLAB functions can provide information

* This work was supported in part by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

[1] MATLAB is a trademark of The MathWorks, Inc.

[2] The use of this initial rank information can be turned off by a compiler flag. In this case, all loaded variables are assumed to be two-dimensional allocatable arrays.

[3] For clarity of presentation, a MATLAB comment (%) is used to indicate which variables are being loaded in the pseudo-code examples presented below. This comment is not required by the translator.

```
S1: n = ...
S2: p = -0.5 * i;
S3: for i=1:n
S4:   p = -0.5 * i;
```

Figure 1: MATLAB pseudo-code segment in which the same expression has different semantics.

```
S1:a = 1:5;              S1:a = 1:5;
S2:for i = 1:5           S2:for i = 1:5
S3: if ( i > 1 )         S3: if ( i == 1 )
S4:  cs(i) = s + a(i);   S4:  s = a(i);
S5:  s = cs(i);          S5:  cs(i) = s;
S6: else                 S6: else
S7:  s = a(i);           S7:  cs(i) = s + a(i);
S8:  cs(i) = s;          S8:  s = cs(i);
S9: end                  S9: end
    ...                      ...
     (a)                      (b)
```

Figure 2: Two MATLAB code segments to compute the cumulative sum of a vector.

for both forward and backward inference. For example, the function `rcond`, for the conditional reciprocal estimator, requires the input parameter to be a square MATRIX. This information is useful for backward inference. Also, the fact that the output of `rcond` is a SCALAR of intrinsic type REAL can be used for forward inference.

Since most operations in MATLAB that have an empty vector as an argument will result in an empty vector, static inference would be seriously hindered if any expression could evaluate to the empty vector. For this reason, we assume that when a variable is used, its value is never the empty vector. However, we allow the assignment of the empty vector to a column or a row of a matrix as long as the assignment does not transform the matrix into an empty vector.

In the rest of this section, we will discuss the main topics on the static inference mechanism, starting with the algorithm used to distinguish between variables and functions, followed by a description of the internal representation, the intrinsic type inference, and the shape and rank inference. For clarity of presentation, we will discuss intrinsic type inference, and shape and rank inference separately, despite the fact that they are applied by a single compiler pass.

### 2.1 Distinguishing Variables from Functions in MATLAB

In MATLAB, an identifier occurrence may represent a function or a variable, depending on the context in which the identifier is first referenced and the files present in the execution environment. For example, consider the pseudo-code presented in Figure 1. The identifier "`i`" in statement S2 is a built-in function that returns the imaginary unit, whereas in statements S3 and S4, "`i`" is a loop index variable.

In general, a MATLAB identifier may represent a function instead of a variable if, in the lexicographic order of the program, it appears on the right hand side (RHS) before it appears on the left hand side (LHS). If the identifier is not a valid built-in function, the files in the execution environment determine whether the identifier represents a function. An example of this is presented in Figure 2, which contains two versions of pseudo-code to compute the cumulative sum of a vector.

The execution of both code segments in Figure 2 will generate the same results if there is no M-file[4] "`s.m`" defined in the user's path or in MATLAB's path. However, code segment (a) will generate wrong results (or may generate an error) if such a file exists. If a function is not found in any of the paths, MATLAB considers the identifier as a variable. Therefore, as the first reference to the identifier "`s`" in code segment (a) appears on the RHS, MATLAB will execute the program correctly only if there is no "`s.m`" M-file. The code segment in Figure 2(b) will always generate correct results because the first reference to the variable "`s`" appears on the LHS.

In order to differentiate variables from functions during the compilation of MATLAB programs, we require that the translation take place in the same environment as the execution. We use the following algorithm, which simulates MATLAB's behavior exactly, to distinguish between variables and functions.

**Algorithm**: *Differentiation Between Variables and Functions*

*Input:* A MATLAB program and its correspondent symbol table.
*Output:* The *kind* of each identifier in the program. That is, whether the identifier represents a variable, a function, or both throughout the program.

This algorithm is based on the state diagram presented in Figure 3. Each identifier can be in one of the following states: Built-in, M-file, Empty, Function, Variable, or Multiple. The Start state is responsible for the definition of the initial state for each identifier. This definition is based on one of the following three conditions:

1. An identifier will start in the Built-in state if it is in the pre-defined list of valid MATLAB built-in functions.
2. An identifier `k` will start in the M-file state if there is an M-file `k.m` in the user's path or in MATLAB's path.
3. If the identifier is neither a Built-in nor an M-file, then it starts in the Empty state.

After defining the initial state of all identifiers, the MATLAB program is traversed in lexicographic order. Every identifier occurrence (*read* or *write*) causes a transition in the state diagram. The state at the end of the program is the kind of the identifier. An identifier will be in the state Multiple if at some point in the program it represented a function (Built-in or M-file) and became a variable later in the program. An example of an identifier of the kind Multiple is "`i`" in Figure 1.

### 2.2 Internal Representation

Our static inference algorithms are applied to a Static Single Assignment (SSA) representation [7] of the MATLAB program. In the SSA representation, each variable is assigned a value by at most one statement. When several definitions feed a single use of a variable, one or more $\phi$ function operators are inserted at the points of confluence in the control flow graph to merge the different definitions into a single variable. SSA is a convenient representation for our analysis algorithms because it is evident which definitions affect (or cover) a particular use of a scalar variable. Because the only control structures in MATLAB are IF statements and loops (the language does not include "`goto`" statements), the SSA representation of a MATLAB program is very simple. In fact, the $\phi$ functions are always at the confluence of two control arcs, which means $\phi$ functions always have two parameters. Each parameter can have a *backward definition* if it corresponds to a variable previously defined in lexicographic order, or a *forward definition*

[4] M-files are user-written functions consisting of a sequence of standard MATLAB statements that possibly include references to other M-files.

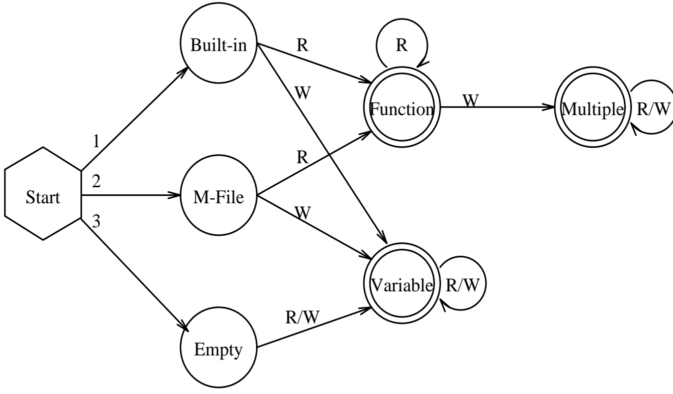


Figure 3: State diagram for differentiation between functions and variables.

otherwise. We call the $\phi$ functions at the end of IF statements $\lambda$ functions, and those at the beginning of a loop $\varphi$ functions. Notice that in $\lambda$ functions, both parameters have backward definitions; in $\varphi$ functions, one parameter has a forward definition and the other a backward definition.

#### 2.2.1 Extension of the SSA Representation to Support Indexed Assignments

Whereas renaming variables for single assignment is easy in the case of scalars and full array assignments, indexed array assignments require a more complex approach involving a new function. Consider, for example, the MATLAB code segment presented in Figure 4, and assume that `A` is assigned in statements S2 and S4 only. If statement S4 did not involve subscripts and were of the form `A = Z`, the SSA representation would be obtained by simply renaming `A` in S2 and S4 and renaming the corresponding uses. However, since S4 updates only a part of `A`, a simple renaming of `A` would not be correct. In situations like this, we use the standard approach and transform indexed assignments of the form `A(R) = ...;`, where `R` is an arbitrary range, into

$$A_{i+1} = \alpha(A_i, R, RHS);$$

The $\alpha$ function we use is similar to the "**Update**" function described in [7], but extended for assignments to a matrix range. In the case of MATLAB, an $\alpha$ function may return an array with dimensions larger than that of the parameter array. In this case, if the array section `R` completely covers the old range of `A`, the values and shape of `A` are those of the RHS. Otherwise, array `A` will have to be expanded in one or both dimensions [5], and the unassigned elements are given the value zero. Using this extension, the SSA representation for the code presented in Figure 4 will be as shown in Figure 5, where the assignments to `A` have been renamed and merged in the same manner as for scalar assignments.

### 2.3 Intrinsic Type Inference

Although MATLAB operates on REAL and COMPLEX values only, our inference mechanism also considers INTEGER and LOGICAL values. That is, it considers all Fortran 90 intrinsic data types except for CHARACTER.

The static intrinsic type inference mechanism propagates types through expressions using a *type algebra* similar to that described in [19] for SETL. Tables containing for each operation the type of the result as a function of the type of the operands are used to

[5] All matrices in MATLAB have lower dimension 1. Hence, only the upper dimension of a range is considered for matrix expansions.

```
S0: load %(n,m)
S1: Z = zeros(n,m);
S2: A = rand(n,m);
    ...
S3: while (...)
    ...
S4:  A(i:n,j:m) = Z(i:n,j:m);
    ...
S5: end
    ...
```

Figure 4: MATLAB code with indexed array assignment

```
S0: load %(n,m)
S1: Z1 = zeros(n1,m1);
S2: A1 = rand(n1,m1);
    ...
S3: while (...)
S4:  A3 = φ(A2, A1);
    ...
S5:  A2 = α(A3, (i1:n1,j1:m1), Z1(i1:n1,j1:m1));
    ...
S6: end
S7: A4 = λ(A1, A2);
    ...
```

Figure 5: SSA representation for indexed array assignments.

| Type of first parameter | | Type of second parameter | | | | | |
|---|---|---|---|---|---|---|---|
| | | **Ø** | **L** | **I** | **R** | **C** | **?** |
| EMPTY | **(Ø)** | **Ø** | **L** | **I** | **R** | **C** | **?** |
| LOGICAL | **(L)** | **L** | **L** | **I** | **R** | **?** | **?** |
| INTEGER | **(I)** | **I** | **I** | **I** | **R** | **?** | **?** |
| REAL | **(R)** | **R** | **R** | **R** | **R** | **?** | **?** |
| COMPLEX | **(C)** | **C** | **?** | **?** | **?** | **C** | **?** |
| UNKNOWN | **(?)** | **?** | **?** | **?** | **?** | **?** | **?** |

Table 1: Type of $\lambda$ and $\varphi$ as a function of their parameters.

implement this algebra. In some cases, the type of an expression depends on the values of the operands. For example, the computation of $\sqrt{i-j}$ may result in INTEGER, REAL, or COMPLEX output, depending on the values of $i$ and $j$. To deal with these cases, we apply a value propagation analysis to keep track of the minimum and maximum values of the variables at each point in the program. For each assignment, we infer the range for the output value by performing the operation using the minimum and maximum values of the operands. Whenever these values cannot be inferred statically, the minimum and maximum values are assigned to $-\infty$ and $+\infty$ respectively, and the static type inference promotes the output of the expression to be of a type that subsumes all possible output types.

If the intrinsic type can be determined statically, a Fortran declaration is generated. Otherwise, if an expression is inferred to be of UNKNOWN type, then the assignment corresponding to the AST node is left to be resolved at execution time.

The propagation of type information requires some consideration in the presence of $\lambda$, $\varphi$, and $\alpha$ functions. In the case of $\lambda$ functions, both parameters have backward definitions. Hence, when the inference system reaches a $\lambda$ function, the type inference system simply compares the types of parameters and assigns the output type according to Table 1.

The static inference of a $\varphi$ function is also based on Table 1; however, the inference mechanism must iterate because the forward definition has yet to be analyzed when it is encountered for the first time. To this end, the forward definition is assumed to be EMPTY

| RHS Type | | Type of previous definition | | | | | |
|---|---|---|---|---|---|---|---|
| | | **Ø** | **L** | **I** | **R** | **C** | **?** |
| EMPTY | **(Ø)** | **Ø** | **L** | **I** | **R** | **C** | **?** |
| LOGICAL | **(L)** | **L** | **L** | **I** | **R** | **C** | **?** |
| INTEGER | **(I)** | **I** | **I** | **I** | **R** | **C** | **?** |
| REAL | **(R)** | **R** | **R** | **R** | **R** | **C** | **?** |
| COMPLEX | **(C)** | **C** | **C** | **C** | **C** | **C** | **C** |
| UNKNOWN | **(?)** | **?** | **?** | **?** | **?** | **C** | **?** |

Table 2: Resulting types for $\alpha$ functions.

the first time, and the inference mechanism iterates to a fixed point. Note that because of the assumption that no variable may have the empty vector as a value when it is used, for $\lambda$, $\varphi$, and $\alpha$ functions, combining an EMPTY type with any other non-EMPTY type results in the non-EMPTY type. Notice also that in Table 1, the confluence of REAL and COMPLEX is UNKNOWN rather than COMPLEX. In this way, if a variable can contain both REAL and COMPLEX values at execution time, complex operations will be executed only when the variable has a complex value. Whether this is better than assigning COMPLEX intrinsic type to the variable is still an open question.

Finally, in the case of $\alpha$ functions, the type inference mechanism compares the type of the RHS with that of the previous definition of the variable being assigned. The resulting type is computed using Table 2. As can be observed in this table, to avoid the overhead of copying the array at execution time, whenever the RHS type differs from that of the previous definition, we promote the resulting $\alpha$ function type to one that subsumes both types. Thus, if an $\alpha$ function output is inferred to be of a more general type than its previous definition, then the intrinsic type of the previous definition is set with this more general type and a backward inference process is activated to update the previous definition and the subsequent variables that use it.

### 2.4 Shape and Rank Inference

Shape inference is very important for the efficiency of the code. If the dimensions of the variables are known during compile time, they can be statically declared and the overhead of dynamic allocation can be avoided. However, in some cases, it is impossible to statically determine the shape of a variable as it can depend on the input data. Thus, the only alternative is to dynamically allocate the variable. In this case, as discussed in Section 3, we try to predict the maximum size of an array in order to avoid the overhead of multiple execution time tests and reallocations.

Rank inference could be avoided by assuming, as MATLAB does, that all variables (even scalars) are matrices. However, this approach has a negative impact on performance because the unnecessary use of indices and the over-dimensioning of variables will cause poor memory and cache utilization. Therefore, it is important to have the capability to recognize scalars and vectors.

For each variable, the outcome of the static inference mechanism is one of the following:

*Exact rank*: When all dimensions are known, the result is one of the following ranks: MATRIX, ROWVECTOR, COLUMNVECTOR, or SCALAR.

*Exclusive rank*: When only one dimension is known, if the known dimension is 1, the variable is inferred to have a NOTMATRIX rank. Otherwise, it is inferred to have a NOTSCALAR rank.

*Unknown rank*: Variables for which the static inference mechanism cannot infer any of the dimensions are considered to have UNKNOWN rank.

| A ∗ B | B | | | |
|---|---|---|---|---|
| A | **S** | **V**(p,1) | **V**(1,q) | **M**(p,n) |
| SCALAR | **S** | **V**(p,1) | **V**(1,q) | **M**(p,n) |
| VECTOR(p,1) | **V**(p,1) | error | **M**(p,q) | error |
| VECTOR(1,q) | **V**(1,q) | **S**[1] | error | **V**(1,n)[1] |
| MATRIX(m,q) | **M**(m,q) | **V**(m,1)[1] | error | **M**(m,n)[1] |

[1] Only if p = q; otherwise error.

Table 3: Exact rank and shape inference for the multiplication operator.

```
S1: if (A_D1 .ne. B_D1 .or.  &
        A_D2 .ne. B_D2) then
S2:  if (ALLOCATED(A)) DEALLOCATE(A)
S3:  A_D1 = B_D1
S4:  A_D2 = B_D2
S5:  ALLOCATE(A(A_D1,A_D2))
S6: end if
S7: A = B + 0.5
```

Figure 6: Example of shadow variables for shape.

Although compiled languages, such as C and Fortran, do not differentiate between row vectors and column vectors, we need to make this distinction because in MATLAB the semantics of some operators are different in each case. Variables that are not inferred to have an exact rank are assumed to be two-dimensional allocatable arrays.

The algorithms for propagation of rank and shape information are very similar to the algorithms for intrinsic type inference, as described in Section 2.3. In fact, rank and shape inference are integrated with intrinsic type inference and are implemented in the same compiler pass. The algebra used by the rank and shape inference algorithm is illustrated for the product operator in Table 3. In this table, the shape information is represented with letters (m, n, p, and q), and ROWVECTORS and COLUMNVECTORS are represented as VECTOR(p,1) and VECTOR(1,q) respectively. During this phase of the analysis, the shape information is obtained only from constants. A symbolic analysis for shape inference is performed during the dynamic inference phase, as described in Section 3.

## 3 Dynamic Inference Mechanism

When the static inference mechanism is unable to infer some properties of a particular variable, the translator generates code to make the inference at execution time. We follow an approach similar to that used in many systems: associating tags with each variable of UNKNOWN intrinsic type or shape. Based on these tags, which are stored in *shadow variables*, conditional statements are used to select the appropriate operations based on the intrinsic type, and to allocate arrays based on the dynamically computed shape. For example, if the shape of `B` in an assignment of the form `A=B+0.5` were UNKNOWN at compile time, the system would generate the code in Figure 6, where shadow variables `A_D1`, `A_D2`, `B_D1`, and `B_D2` are used to store the execution time information about the dimensions of `A` and `B`. Similar shadow variables are generated for dynamic intrinsic type inference.

One problem with this approach is that the number of possible type combinations grows exponentially with the number of operands. To reduce this problem, the dynamic analysis system for intrinsic type considers only REAL and COMPLEX types, and all expressions with more than two operands are transformed into a sequence of *triplets* using three-address code in the form: $\mathtt{t} \leftarrow \mathtt{x}$ OP $\mathtt{y}$. Hence, for an expression with $n$ operands, there

```
S1: for k=1:n
S2:   P(k,k)=4;
S3: end
S4: for j=1:n-1
S5:   P(j,j+1)=-1;
S6:   P(j+1,j)=-1;
S7: end
```

Figure 7: Part of a MATLAB code to generate a Poisson matrix.

```
    if (k .gt. P_D1 .or. k .gt. P_D2) then
      if (ALLOCATED(P)) then
        T0_D1 = P_D1
        T0_D2 = P_D2
        ALLOCATE(T0_R(T0_D1, T0_D2))
        T0_R = P
        DEALLOCATE(P)
        P_D1 = MAX(P_D1, k)
        P_D2 = MAX(P_D2, k)
        ALLOCATE(P(P_D1, P_D2))
        P(1:T0_D1, 1:T0_D2) = T0_R
        P(1:T0_D1, T0_D2+1:P_D2) = 0.
        P(T0_D1+1:P_D1, :)  = 0.
        DEALLOCATE(T0_R)
      else
        P_D1 = k
        P_D2 = k
        ALLOCATE(P(P_D1, P_D2))
        P = 0.
      end if
    else
      if (.not. ALLOCATED(P)) then
        P_D1 = k
        P_D2 = k
        ALLOCATE(P(P_D1, P_D2))
        P = 0.
      end if
    end if
S2: P(k , k) = 4
```

Figure 8: Fortran 90 allocation test for the MATLAB expression `P(k,k)=4`.

will be at most $n-1$ triplets with at most 4 cases each ( [REAL,REAL]; [COMPLEX,REAL]; [REAL,COMPLEX]; and [COMPLEX,COMPLEX] ). That is, we consider at most $4(n-1)$ cases as opposed to the $2^n$ cases that would arise if the expression were not decomposed.

### 3.1 Support for Dynamic Shape Inference

Using shadow variables makes the generation of dynamic code straightforward. However, in the case of shape inference, some optimizations are necessary to avoid the excessive number of tests and allocations. To illustrate this, consider the code segment of Figure 7, which computes the tridiagonal part of a Poisson matrix. Let us assume that the value of `n` is not known at compile time. The program resulting from a naive compilation of this code would contain allocation tests just before statements S2, S5, and S6. For example, the allocation test corresponding to statement S2 would be as shown in Figure 8.

The allocation tests before S2, S5, and S6 can be avoided if an allocation of `P` with shape `n`×`n` were placed before statement S1. Avoiding the allocation tests is particularly important if there is no definition of `P` before S1. In fact, if `P` were defined before S1, each allocation test will cause only a small overhead from the conditional statements. On the other hand, if `P` is first referenced in statement S2, loop S1 will produce a very large overhead because `P` will have to be reallocated at each iteration.

This simple example illustrates two static techniques needed to support dynamic shape inference: *use-definition coverage analysis* and *efficient placement of dynamic allocation*. The objective of the first technique is to determine whether an indexed array assignment may increase the size of the array. If this information is known at compile time, it is not necessary to generate an allocation test for the indexed assignment. Otherwise, the allocation test must be generated and the second technique is used to place the test where it will minimize the overhead.

To determine whether there is definition coverage we use a *dimension propagation algorithm* with symbolic capabilities. This algorithm is similar to the range propagation algorithm used by Blume and Eigenmann for the Range Test [3]. Since all matrices in MATLAB have lower dimensions set to 1, our problem is simplified to determining whether the maximum value that an array index will reach is larger than the corresponding dimension in the previous assignment of the variable. If it is larger, then reallocating the array is necessary; otherwise, no dynamic test is necessary for the assignment being considered.

Our dimension propagation algorithm symbolically computes the maximum value of each scalar subscript expression which can be used to generate `ALLOCATE` statements. Consider, for example, the code presented in Figure 7. From the scalar assignment in S1, the compiler determines that the maximum value of the variable `k` will be `n`. Using this information, the compiler determines that S2 creates a matrix `P` with shape `n`×`n`. From S4, the compiler determines that the maximum value of `j` will be `n-1`. Notice that, by using simple symbolic algebra capabilities, it is possible to determine that `j+1` is ≤ `n` and that the shape of P is not expanded in S5 or S6. Therefore, it is not necessary to generate dynamic allocation tests for these statements. All this information is obtained by tracing the indices of `P` to their previous definitions in the SSA representation of the program, following an on-demand approach similar to that introduced in [20]. In some situations (such as non-scalar indices or indirections), this symbolic inference is unable to determine the maximum value. In this case, the compiler sets the corresponding information as UNKNOWN, and dynamic allocation is required.

The technique for the placement of dynamic allocation traces the SSA representation from a variable assignment to its previous definition (D) and to the definition of the variable that determines the symbolic dimension (S)[6]. If there is no previous definition for the assigned variable, the `ALLOCATE` can be placed at any point between S and the assignment. Otherwise, if an allocation test is required after D, it can be placed after both S and the last use of D. In both cases, if possible, the dynamic allocation is placed outside of the outermost loop, to avoid overhead.

Rank and shape can be propagated through matrix constructors, through built-in functions that construct matrices based on the parameters (e.g. `rand`, `zeros`), through built-in functions that return matrices with the same shape as the input parameter (e.g. `sin`, `sqrt`), and through expressions in general. However, this propagation is not always straightforward. Consider, for example, the multiplication `C = A * B`, where `A` and `B` are `n`×`q` and `p`×`m` respectively. We cannot infer that `C` will be an `n`×`m` matrix without knowing the actual values of `n`, `q`, `p`, and `m`. If `n` and `q` are both equal to 1, then `C` will be a `p`×`m` matrix. Similarly, if `p` and `m` are both equal to 1, then `C` will be a `n`×`q` matrix. Thus, without further information about `n`, `q`, `p`, and `m`, it is impossible to infer the symbolic information for the variable `C`.

[6] S refers to the last definition, in lexicographic order, in which two variables define the symbolic dimension of an array.

| Problem | | From | Problem size |
|---|---|---|---|
| Adaptive Quadrature using Simpson's rule | (AQ) | a | 1 Dim. (7) |
| Conjugate Gradient method | (CG) | b | 420 × 420 |
| Generation of a 3D-surface | (3D) | c | 41×21×11 |
| Dirichlet Method for Laplace's Equation | (Di) | a | 40 × 40 |
| Finite difference solution to the wave equation | (FD) | a | 451 × 451 |
| Sources: a - From [17]; b - From [2]; c - Colleagues | | | |

Table 4: Test programs.

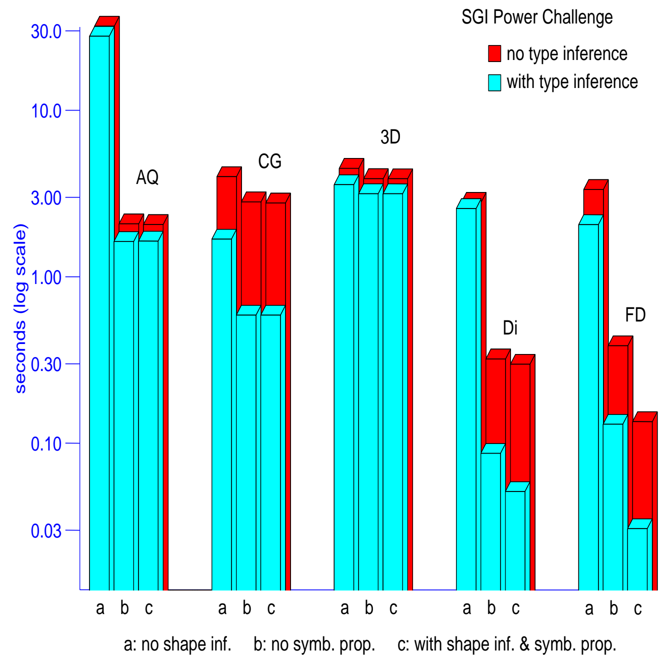


Figure 9: Inference phases comparison.

## 4 Experimental Results

To measure both the effectiveness of the internal phases of the inference mechanism and the overall effectiveness of the compiler, we performed two sets of experiments. In the first set, we measured the importance of each of the major phases of the inference system. In the second set, we compared the performance of compiled codes with their interpreted MATLAB versions and with Fortran 90 hand-written versions of the same algorithms. For both sets of experiments, we ran five MATLAB programs on an SGI Power Challenge using one processor. To avoid large execution times, especially in MATLAB, the time required for experiments was controlled by setting the problem size[7] and the numerical resolution (in the case of iterative problems).

A brief description of the test programs is presented in Table 4. These programs can be classified into three groups, depending upon certain characteristics of the MATLAB code and its execution. CG and 3D spend most of their time executing built-in functions. AQ is placed in the group that requires dynamic reallocation of matrices during execution time. Finally, Di and FD are loop-based programs requiring element-wise access of arrays. A particular characteristic of CG is that it has no indexed array assignments.

### 4.1 Evaluation of the Inference Phases

With the use of compiler flags, we independently deactivated intrinsic type inference, shape and rank inference, and symbolic dimension propagation. When type inference was deactivated, all variables, with the exception of `do` loop indices and temporaries used for conditional statements, were declared COMPLEX. When shape and rank inference were deactivated, all variables, with the same two exceptions just mentioned, were declared as two-dimensional allocatable arrays. For all runs where shape and rank inference were deactivated, dimension propagation was also deactivated since it is an optimization of shape and rank inference. Figure 9 presents the execution times in seconds for each of the programs [8] running with all six possible combinations.

We observe that 3D is the program having the least variation in performance between the different inference phases. This behavior results from the fact that this program spends most of its time executing a library function to calculate eigenvalues. Furthermore, 3D uses a COMPLEX array during this computation, thus, minimizing the effect of intrinsic type inference. Its overall improvement in performance from no inference to all phases being used was on the order of 25%. For all other programs, at least one of the inference phases produced a significant performance improvement.

Shape and rank inference had a large influence on performance for all other programs, with improvements ranging from 3 times faster for CG to almost 30 times faster for Di. The main reason for this improvement is the reduction in overhead for dynamic shape inference, especially for scalars. When dynamic shape inference is necessary for a matrix, the overhead generated by the compiler may be amortized by the subsequent operation and assignment to the matrix, depending on its size and the number of floating point operations performed by the expression. On the other hand, a typical scalar assignment doesn't require enough work to compensate the overhead.

As expected, only the loop based programs requiring element-wise access of arrays (Di and FD) benefited from the dimension propagation. AQ requires reallocation of arrays; hence, the dimension propagation has no effect on the program. Since CG has no indexed assignments and spends most of its time computing library functions, dimension propagation also has a very small effect on the generated program.

Type inference generated the biggest improvements for the computational bounded problems (CG, Di, and FD). In these cases, when shape inference and dimension propagation were activated, the speedup resulting from type inference ranged from 3.8 to 5. On the other hand, type inference had very little effect on AQ since it spends most of its time performing data movements.

### 4.2 Comparison to MATLAB and hand-written Fortran 90 programs

Our experimental results, presented in Table 5, show that for all five programs the performance of the compiled code is better than the respective interpreted execution, and the range of speedups is heavily dependent on the characteristics of the MATLAB program. Programs in the first group have a relatively small speedup compared to MATLAB, since they spend most of their execution time performing the same library functions. The speedup obtained is primarily attributed to the overhead of interpretation.

The speedup obtained by AQ resulted from the better handling of indexed assignments and the reallocation of matrices by the compiled program. However, according to preliminary experiments to

[7] A more detailed study is underway to determine how problem size affects the relative speed of the programs.

[8] For each individual test, the time presented in the graph represents the best time out of five runs.

| Program | MATLAB | Compiled | Speedup | Hand coded |
|---|---|---|---|---|
| AQ | 19.98 | 1.637 | 12.2 | 0.875 |
| CG | 5.30 | 0.589 | 9.0 | 0.542 |
| 3D | 36.04 | 3.161 | 11.4 | 3.130 |
| Di | 44.78 | 0.052 | 861.1 | 0.050 |
| FD | 31.82 | 0.031 | 1026.5 | 0.031 |

Table 5: Execution times (in seconds) running on the SGI Power Challenge.

determine how problem size affects the relative speed of the programs, this improvement varies considerably, depending upon the number of reallocations required by the program, which is in turn dependent upon the input data set and the function being used for the numerical integration. Finally, loop-based programs requiring element-wise access of arrays are the programs that benefit the most from compilation. due to the more efficient loop control structure of the compiled code and the larger overhead of the indexed assignments within the interpreted code.

When comparing the hand-coded Fortran 90 programs with the compiler generated versions, we observe that for these programs the performance of the compiled versions is very close to the performance of the hand-written programs. The largest performance difference occurs with AQ, primarily because of the reallocation of matrices by FALCON, as presented in Figure 8. In the hand-written code, due to a better knowledge of the algorithm, we can avoid part of the copy of the old values to the expanded matrix, and the initialization of the expanded part to zero.

## 5 Related Work

In this section, we will briefly discuss a few other approaches for the compilation of MATLAB or MATLAB-like languages. These approaches range from research projects to commercial products.

CONLAB [13] is an interactive environment for developing algorithms for parallel computer architectures. It uses a subset of the MATLAB language, with extensions for expressing parallelism, synchronization, and communication. A translator from CONLAB to C was developed by Drakenberg et al. [10]. Some sort of type inference system is alluded to by the authors in their papers, but it is not described.

A simple approach for the translation of MATLAB programs into C++ is presented in [14]. In this work, a matrix class was created to take care of all type and shape inference decisions during execution time. This class is then utilized by the generated C++. Effectively, the control structure is compiled, but all the mathematical operations are still interpreted within this matrix class.

Recently, MathWorks released MCC, a MATLAB compiler [16] that translates MATLAB programs into C for stand-alone external applications, or into C MEX-files[9], which are called within the MATLAB environment. From [16], it appears that MCC performs only simple inference and relies upon user-provided flags, pragmas, and assertions, that specify the type or rank of variables, to optimize the generated code. Using the MATLAB programs described above, we compared the performance of C MEX-files generated by MCC[10] and the programs briefly described in Table 6 with the performance of Fortran 90 programs generated by FALCON, on a SPARCstation 10. Due to the lack of a native Fortran 90 compiler on our SPARCstation, the Fortran 90 programs were first translated to Fortran 77 with the use of VAST-90 [18], and then compiled

[9] MEX-files are MATLAB-callable C or Fortran dynamically linked subroutines that are built with a special interface module.

[10] At this time we have only the MathWorks compiler available on a SPARCstation. We are in the process of obtaining the necessary software to perform these experiments on an SGI Power Challenge.

| Problem | | From | Problem size |
|---|---|---|---|
| Successive Overrelaxation method | (SOR) | b | 420 × 420 |
| Quasi-Minimal Residual method | (QMR) | b | 420 × 420 |
| Crank-Nicholson solution to the heat equation | (CN) | a | 321 × 321 |
| Two body problem using 4th order Runge-Kutta | (RK) | d | 3200 steps |
| Two body problem using Euler-Cromer method | (EC) | d | 6240 steps |
| Sources: a - From [17]; b - From [2]; d - From [11] | | | |

Table 6: Additional test programs.

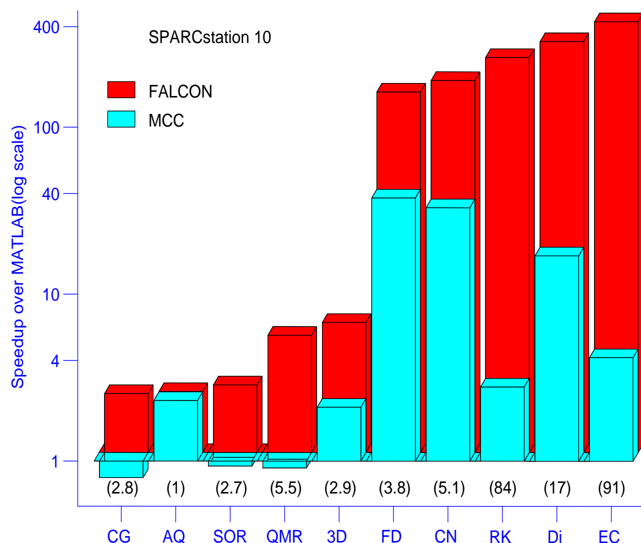


Figure 10: Speedup comparisons. FALCON's speedup over MCC in parenthesis.

with the Sun Fortran 77 compiler using the optimization flag "O3". The MEX-files generated by MCC were compiled with the GNU C compiler using the optimization flag "O3". MCC does not support the `load` statement; hence, the input data was loaded using interpreted MATLAB commands (not timed) and provided to the MEX-files as function parameters. To provide MCC with the same information that was extracted by FALCON from the loaded variables, assertions were added to the M-files.

Figure 10 presents the speedups of the codes generated by both compilers over MATLAB. It also includes, in parenthesis, the speedup of FALCON's generated codes over MCC's codes. With the exception of AQ that, as described previously, spends most of its time performing reallocations and data movements, all other codes generated by FALCON ran at least 2.7 times faster than their MCC counterparts. We observe that in three cases (CG, SOR, and QMR), MCC generated a program that ran slower than MATLAB.

Three programs generated by FALCON (RK, EC, and Di) had significantly better performance than the corresponding MCC versions. RK and EC perform several elementary matrix operations on $2 \times 2$ matrices. The code generated by MCC appears to be very inefficient for these kinds of operations. Furthermore, the lack of "preallocation" of variables in the MATLAB code is also responsible for the degradation of the performance of these MCC codes whereas because of our dimension propagation, FALCON is able to allocate all matrices outside the main loop. Finally, the better

performance from Di results from our value propagation analysis. In this case, the type inference can determine that the expression:

$$\sqrt{4 - \left[cos\left(\frac{\pi}{n-1}\right) + cos\left(\frac{\pi}{m-1}\right)\right]^2}$$

will always return a REAL value, whereas MCC assumes the output of a square root to be of type COMPLEX. Thus, the code generated by MCC for Di uses COMPLEX variables for most of its computations, whereas ours uses only REAL variables.

## 6 Conclusions

We are building a programming environment for the development of scientific libraries and applications. By using MATLAB as the source language and producing Fortran 90 as output, this environment takes advantage of both the power of interactive array languages and the performance of compiled languages. In order to generate code from the interactive array language, we developed a translator that combines static and dynamic inference methods for type, shape, and rank inference, and is optimized with value and symbolic dimension propagation.

As shown by our experimental results, the compiled programs performed better than their respective interpreted executions, with performance improvement factors varying according to the characteristics of each program. For certain classes of programs, our translator generates code that executes as fast as hand-written Fortran 90 programs, and more than 1000 times faster than the corresponding MATLAB execution.

Further performance improvements will be possible by integrating FALCON with a parallelizing compiler, as planned for the second phase of this project. In order to facilitate the work of the parallelizer, we will include in this integration the generation of directives for parallelization and data distribution through the exploitation of the high-level semantics of the array language.

## References


[1] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1985.

[2] BARRETT, R., BERRY, M., CHAN, T., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND VAN DER VORST, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1993.

[3] BLUME, W., AND EIGENMANN, R. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. In *Proceedings of Supercomputing '94* (November 1994), pp. 528–537.

[4] BLUME, W., EIGENMANN, R., FAIGIN, K., GROUT, J., HOEFLINGER, J., PADUA, D., PETERSEN, P., POTTENGER, B., RAUCHWERGER, L., TU, P., AND WEATHERFORD, S. Polaris: Improving the Effectiveness of Parallelizing Compilers. In *Languages and Compilers for Parallel Computing* (August 1994), K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Lecture Notes in Computer Science, vol. 892, Springer-Verlag, pp. 141–154. 7th International Workshop, Ithaca, NY, USA.

[5] BUDD, T. *An APL Compiler*. Springer-Verlag, 1988.

[6] CHING, W.-M. Program Analysis and Code Generation in an APL/370 Compiler. *IBM Journal of Research and Development 30:6* (November 1986), 594–602.

[7] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems 13*, 4 (October 1991), 451–490.

[8] DEROSE, L., GALLIVAN, K., GALLOPOULOS, E., MARSOLF, B., AND PADUA, D. FALCON: A MATLAB Interactive Restructuring Compiler. In *Languages and Compilers for Parallel Computing* (August 1995), C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Lecture Notes in Computer Science, vol. 1033, Springer-Verlag, pp. 269–288. 8th International Workshop, Columbus, Ohio.

[9] DEROSE, L., GALLIVAN, K., GALLOPOULOS, E., MARSOLF, B., AND PADUA, D. FALCON: An Environment for the Development of Scientific Libraries and Applications. In *Proc. of the KBUP95: First international workshop on Knowledge-Based systems for the (re)Use of Program libraries* (Sophia Antipolis, France, November 1995).

[10] DRAKENBERG, P., JACOBSON, P., AND KAGSTROM, B. A CONLAB Compiler for a Distributed Memory Multicomputer. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, Norfolk Va* (March 1993).

[11] GARCIA, A. L. *Numerical Methods for Physics*. Prentice Hall, 1994.

[12] GILMAN, L., AND ROSE, A. *APL : An Interactive Approach*. Wiley, 1984.

[13] JACOBSON, P., KAGSTROM, B., AND RANNAR, M. Algorithm Development for Distributed Memory Multicomputers Using CONLAB. *Scientific Programming 1* (1992), 185–203.

[14] KEREN, Y. *MATCOM: A MATLAB to C++ Translator and Support Libraries*. Technion, Israel Institute of Technology, 1995.

[15] THE MATH WORKS, INC. *MATLAB, High-Performance Numeric Computation and Visualization Software. User's Guide*, 1992.

[16] THE MATH WORKS, INC. *MATLAB Compiler*, 1995.

[17] MATHEWS, J. H. *Numerical Methods for Mathematics, Science and Engineering*, 2nd ed. Prentice Hall, 1992.

[18] PACIFIC-SIERRA RESEARCH CORPORATION. *VAST-90 Fortran 90 Language System: User guide*, 2.1 ed., 1993.

[19] SCHWARTZ, J. T. Automatic Data Structure Choice in a Language of a Very High Level. *Communications of the ACM 18* (1975), 722–728.

[20] TU, P., AND PADUA, D. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing* (Barcelona, Spain, July 1995), pp. 414–423.