Techniques for the Translation of MATLAB Programs into Fortran 90

LUIZ DE ROSE and DAVID PADUA
University of Illinois at Urbana-Champaign

This article describes the main techniques developed for FALCON's MATLAB-to-Fortran 90 compiler. FALCON is a programming environment for the development of high-performance scientific programs. It combines static and dynamic inference methods to translate MATLAB programs into Fortran 90. The static inference is supported with advanced value propagation techniques and symbolic algorithms for subscript analysis. Experiments show that FALCON's MATLAB translator can generate code that performs more than 1000 times faster than the interpreted version of MATLAB and substantially faster than commercially available MATLAB compilers on one processor of an SGI Power Challenge. Furthermore, in most cases we have tested, the compiler-generated code is as fast as corresponding hand-written programs.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Very high-level languages*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; I.1.2 [**Symbolic and algebraic manipulation**]: algorithms

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Array language compilation, inference, MATLAB

1. INTRODUCTION

The development of software for scientific computation on high-performance computers is a difficult and time-consuming task. We believe the development of scientific programs should start with a language that corresponds as closely as possible to the mathematical description of the problem, albeit in the form of a simple and easy-to-use procedural language. The use of the appropriate high-level language facilitates the development process by enhancing the ease of programming and portability of applications.

Interactive array languages, such as APL [Gilman and Rose 1984], its modern version J [Iverson 1995], and MATLAB [MathWorks 1992], are powerful tools for

This work was supported in part by NSF contract ACI-9870687; Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; and National Computational Science Alliance grant number ASC980005N. This work is not necessarily representative of the positions or policies of the Army or Government.
Authors' addresses: Luiz De Rose, Advanced Computing Technology Center, IBM T.J. Watson Research Center, P.O. Box 218, Route 134/Kitchawan Road, Yorktown Heights, NY 10598; email: laderose@us.ibm.com; David Padua, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Av., Urbana, IL 61801; email: padua@cs.uiuc.edu.

developing programs for numerical computation. A convenient feature in these languages that facilitates prototyping of applications is that declaring dimensions and intrinsic type of variables is unnecessary.[1] Moreover, interactive array languages are usually contained within environments that include easy-to-use facilities for displaying results both graphically and in tabular form [Gallopoulos et al. 1994]. Furthermore, the interactive nature of these languages provides an environment that tends to increase productivity in software development. The trade-off is that in order to provide such a convenient programming environment, array languages are usually interpreted, which results in negative effects on performance.

To implement scientific codes a programmer could develop a prototype in an interactive language, like MATLAB, and then rewrite it in a compiled language, like C or Fortran. In practice, however, the overhead of reimplementing programs in a different language is large enough that most people seldom follow through with this option. Clearly the best solution is for programmers to use a translator that directly generates efficient code from MATLAB programs.

We have implemented a MATLAB-to-Fortran 90 source-to-source translator in FALCON [DeRose et al. 1995a; 1995b], a programming environment for the development of scientific libraries and applications. In this article, we discuss the techniques used by this translator to overcome problems, such as the following, in the MATLAB language: the lack of intrinsic type definitions and specification of variables' dimensions; the possibility that any of these variable properties could change during execution time; and, the overloading of operators that have different semantics, depending on the rank of the variables on which they operate. The techniques used to overcome these problems include *static* inference mechanisms used to generate declarations at compile time, and *dynamic* strategies that are applied at run-time when the lack of statically available information prevents the automatic generation of a particular variable's declaration.

Using static inference mechanisms to compile interactive languages is not a new idea. In particular, APL compiler projects [Budd 1988; Ching 1986; Guibas and Wyatt 1978] inspired our work, and MCC [MathWorks 1995], a commercial MATLAB compiler that generates C code, was developed at the same time our translator was being built.

However, the differences between MATLAB and APL are significant enough to justify a separate study. Thus, it is important to evaluate the effectiveness of static and dynamic inference techniques on MATLAB codes because it is not clear that static inference techniques would be equally effective on MATLAB and APL. Also, we found that analysis techniques not very important in APL are necessary in MATLAB programs in order to attain good results. APL compiler techniques focus on inferences at the expression level because APL array manipulation is almost always done by applying powerful operations combined in a single expression. In fact, APL single-line programs, known as one-liners, are frequent and encouraged. On the other hand, in MATLAB, array operators are usually spread across different assign statements, using high-level control structures, like `DO` loops, and complex

[1]For example, MATLAB 4.2 has a single data type, a two-dimensional array of double-precision complex numbers that may grow dynamically. Thus, scalars, vectors, and matrices holding integer, real, or complex values are all stored into the same class of data structure.

control subscript expressions involving triplets. Because of this, accurate inference in MATLAB often requires value propagation to determine the range of values that can be assumed by subscript expressions. Furthermore, in MATLAB, it is important to determine how a particular variable changes in a region of the program. In particular, it is important to determine the maximum size of an array in a loop to avoid the need for run-time checks. The translator presented in this article differs from APL compilers in that it relies more heavily on global propagation and applies loop analysis techniques to identify invariants. Furthermore, this compiler contains advanced computer algebra algorithms that have proven to be of great importance in type inference and value propagation. As far as we could determine, the APL compilers described in the literature [Budd 1988; Ching 1986; Guibas and Wyatt 1978] do not contain such capabilities.

This article is organized as follows. Section 2 presents the techniques needed to identify functions in MATLAB. Section 3 describes the internal representation used by the FALCON compiler. The algorithms for the static inference mechanism are presented in Section 4. The run-time inference algorithms are discussed in Section 5. Experimental results are presented in Section 6. Section 7 presents examples of how some of the techniques described in this article can be used for the compilation of Java programs. Our conclusions are presented in Section 8. Finally, for the reader unfamiliar with MATLAB, we have included an appendix that describes MATLAB and discusses its similarities with APL and Fortran 90.

2. IDENTIFICATION OF MATLAB FUNCTIONS

The FALCON MATLAB compiler performs whole-program analysis on the input MATLAB program. In general, MATLAB programs contain several function calls; thus, interprocedural analysis is necessary in the compiler to define the properties of the output parameters. In the case of built-ins, the compiler uses a database that contains all the necessary information about the function for the inference process. Hence, for a particular built-in, it is possible to retrieve the variable properties of the output parameters given the variable properties of the input arguments. For the case of M-files, we approached the inter-procedural analysis problem by *inlining* all the M-files that are used in the program, including multiple levels of M-file calls (i.e., M-files called inside another M-file). We then analyze the entire program at once. Thus, in order to inline the M-files, we require all M-files that are used in the program to be present in the user's path or in MATLAB's path during the translation.

In MATLAB, an identifier occurrence represents a function or a variable, depending on the context in which the identifier is first referenced and the files present in the execution environment. In general, a MATLAB identifier may represent a function instead of a variable if it appears on the *right-hand side* (RHS) before it appears on the *left-hand side* (LHS). If the identifier is not a built-in function, the files in the execution environment determine whether the identifier represents an M-file.[2] An example of this behavior is presented in Figure 1(a), which contains

[2]An identifier that first appears on the RHS can also be a MEX-file. However, the current version of the FALCON compiler does not support MEX-files, so we do not consider MEX-files in this presentation.

```
      ...
S1:  for iter = 1:max_it
S2:    z = M \ r;
S3:    rho = (r'*z);
S4:    if ( iter > 1 ),
S5:      beta = rho / rho_1;
S6:      p = z + beta*p;
S7:    else
S8:      p = z;
S9:    end
       ...
         (a)
```

```
      ...
S1:  for iter = 1:max_it
S2:    z = M \ r;
S3:    rho = (r'*z);
S4:    if ( iter = 1 ),
S5:      p = z;
S6:    else
S7:      beta = rho / rho_1;
S8:      p = z + beta*p;
S9:    end
         ...
         (b)
```

Fig. 1. Two MATLAB code segments from a Conjugate Gradient solver.

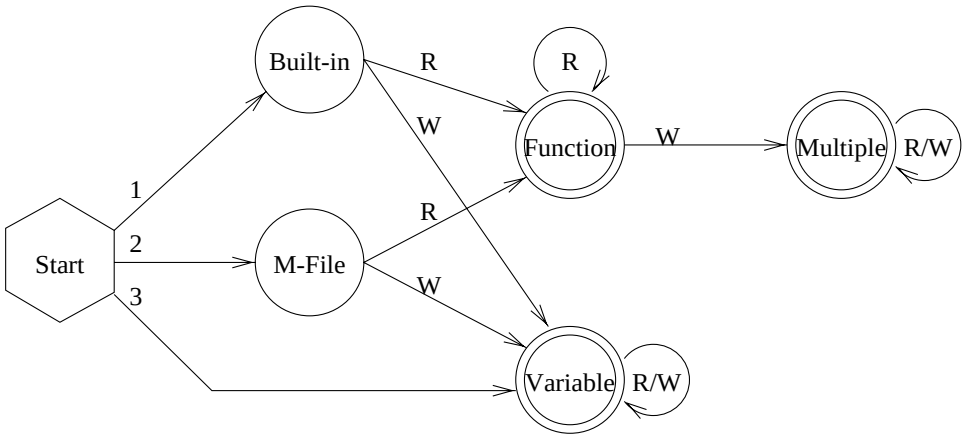

Fig. 2. State diagram for differentiation between functions and variables.

a code segment from Barrett et al. [1993] with a Conjugate Gradient solver, and Figure 1(b), which has a slightly modified code segment for the same algorithm.

The execution of both code segments in Figure 1 will generate the same results if there is no M-file "`p.m`". However, code segment (a) will generate wrong results (or may generate an error) if such a file exists. The reason for this inconsistent behavior is that if MATLAB, while parsing the program, finds an identifier on the RHS without a previous definition on the LHS, it considers the identifier a function and searches for its definition in the user's path and, if not found, in its own path. If a function is not found in any of the paths, MATLAB considers the identifier a variable. Therefore, as the first reference to the identifier "`p`" in code segment (a) appears on the RHS, MATLAB will execute the program correctly only if there is no "`p.m`" M-file. On the other hand, the code segment in Figure 1(b) will always generate the correct result because the first reference to the variable "`p`" appears on the LHS.

In order to differentiate variables from functions during the translation of the MATLAB program, we designed Algorithm 1, which simulates MATLAB's behavior to distinguish between variables and functions.

Algorithm 1. *Differentiation Between Variables and Functions:*

Input: An *abstract syntax tree* (AST) representation of a MATLAB program and its corresponding symbol table.
Output: For each identifier, information on whether the identifier represents a variable, a function, or both throughout the program.

This algorithm is based on the state diagram presented in Figure 2, with one state diagram for each identifier in the program. Each identifier can be in one of the following states: Built-in, M-file, Function, Variable, or Multiple. The initial state for each identifier is based on one of the following three conditions:

(1) if an identifier is in the predefined list of valid MATLAB built-in functions, it will start in the Built-in state; otherwise
(2) if there is an M-file `k.m` in the user's path or in MATLAB's path, an identifier `k` will start in the M-file state;
(3) if the identifier is neither a Built-in nor an M-file, then it is a variable.

After defining the initial state of all identifiers, the AST is traversed in program order. Every identifier occurrence (*read* (R) or *write* (W)) causes a transition in the state diagram. The state at the end of the program tells us whether the identifier is a variable or a function. 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.

3. INTERMEDIATE REPRESENTATION

FALCON's static inference algorithms are applied to a *static single-assignment* (SSA) [Cytron et al. 1991] representation of the MATLAB program. A program in SSA form has two main properties: each scalar variable is assigned a value by at most one statement, and there is only one definition for each use of the variable. When in the original program several definitions feed a single use of a variable, one or more ϕ function operators are inserted at the points of confluence in the control flow graph to merge the different definitions into a single variable.

MATLAB lacks type definitions, and any variable could change type during runtime. SSA, therefore, enables an efficient implementation of our analysis algorithms because in SSA it is evident which definitions affect (or cover) a particular use of a variable. In addition, as opposed to the traditional data-flow framework, the SSA representation deals in an uniform manner with scalars and arrays. Moreover, due to the structured nature of MATLAB (having only `if` statements, structured loops, and no `goto`s), the SSA representation of a MATLAB program is very simple. In fact, the ϕ functions are always at the confluence of exactly two control arcs. As a result, ϕ functions always have two parameters, which facilitates the design and implementation of our algorithms. Each parameter can have a *previous definition* if it corresponds to a variable previously defined in the program, or a *forward definition* if the variable definition follows the ϕ function. A ϕ function parameter having a forward definition will only occur in loops. In this case, the ϕ function appears at the top of the loop, and its second parameter will always have a previous definition.

```
S0: load %(n)

S1: for k=1:n

S2:   if (k > 1)
S3:     z = y / k;
S4:   else
S5:     z = k;
S6:   end

      ...
S7:     y = z;
S8: end
```

(a)

```
S0: load %(n1)
P0: z0 = ∅
P1: y0 = ∅
S1: for k1=1:n1
P2:   y2 = β(y1,y0)
P3:   z4 = β(z3,z0)
S2:   if (k1 > 1)
S3:     z1 = y1 / k1;
S4:   else
S5:     z2 = k1;
S6:   end
P4:   z3 = λ(z1,z2)
      ...
S7:     y1 = z3;
S8: end
P5: y3 = λ(y1,y0)
P6: z5 = λ(z3,z0)
```

(b)

Fig. 3. MATLAB code segment (a) and corresponding SSA representation (b).

As illustrated in Figure 3, which displays a MATLAB code segment and its corresponding SSA representation, our ϕ functions are classified into two groups, according to the kind of construction required by the function. The first group of ϕ functions, which we call λ functions, replaces the ϕ functions required at the end of conditional statements and loops (represented by P4, P5, and P6 in Figure 3(b)). Both parameters of λ functions have previous definitions. The other group of ϕ functions, which we redefine as β functions, replaces the ϕ functions required at the beginning of loops (P2 and P3 in Figure 3(b)). β functions always have one forward definition and one previous definition. Notice that loops require two ϕ functions for each variable in the LHS of the body of the loop. β functions (P2 and P3 in Figure 3(b)) are required at the beginning of the loop body to differentiate between the path coming from the first iteration and the path coming from the subsequent iterations, and λ functions (P5 and P6 in Figure 3(b)) are required after the end of the loop, since the translator cannot always be sure if the loop body is going to be executed or not. If the first occurrence of a variable occurs inside of a loop, the translator creates an artificial assignment to `null` just before the loop to serve as the previous definition (these definitions are represented by P0 and P1 in Figure 3(b)).

The MATLAB language allows *full array assignments* (e.g., `A=B` where `B` is an array), as well as *indexed array assignments* (e.g., `A(J)=RHS`, where `J` is an arbitrary *index* that can be a scalar, a vector, or a two-dimensional array represented with constants, variables, or both). Full array assignments can be treated in the SSA form as scalar assignments, since each assignment completely redefines its previous definition. Whereas it is easy to determine which definitions cover a given use in the case of scalars and full array assignments using the SSA representation, indexed array assignments require a more complex analysis. In our representation, we use the α function, which represents indexed assignments of the form `A(J) = RHS` as

$$\mathtt{A}_{i+1} = \alpha(\mathtt{A}_i, \mathtt{J}, \mathtt{RHS}).$$

This α function is similar to the "**Update**" function described by Cytron et al. [1991]. It assigns to all elements of $\texttt{A}_{i+1}$ the corresponding elements of $\texttt{A}_i$, except for $\texttt{A}_{i+1}\texttt{(J)}$, which is assigned the RHS. The main difference between the α function and the "**Update**" function is that `J` can be a vector or a matrix. Furthermore, one or both dimensions of $\texttt{A}_{i+1}$ may be larger than the corresponding dimensions of $\texttt{A}_i$. In this case, unassigned elements will be given the value zero.

4. STATIC ANALYSIS

The main challenge of the MATLAB translator is to perform inference on the input program to determine the following variable properties necessary to generate the Fortran 90 declarations and to optimize the output code:

—*intrinsic type*, which could be `complex`, `real`[3] `integer`, or `logical`;
—*shape*, which could be `scalar`, `vector`, or `matrix`;[4] and
—*size*, which indicates the size of each dimension.

The static inference mechanism extracts information from four main sources: input files, program constants, operators, and built-in functions. If the program reads data from an external file, the system requires a sample file to be present during the compilation. From this file, the compiler extracts the initial intrinsic type and shape[5] of the variables being loaded. Variable sizes are not extracted from the input files because they are much more likely than intrinsic type and shape to differ between runs. If the type information provided by the sample file differs from the file used during run-time, the generated Fortran 90 program would probably be incorrect. Hence, the compiler places run-time type checks for each variable loaded in the program. If the intrinsic type or shape of any loaded variable differs from the type expected by the compiler, the program will exit with a message indicating the type difference.

If a sample of the input file were not available for compilation, the translator would have to be conservative and assume all input variables to be `complex` two-dimensional arrays. This assumption would hinder the inference process for most programs, as the types of the input variables would be propagated through the program. We do not compile single functions independently for the same reason: the types of the input parameters are necessary for accurate inference. In Section 6.4, we discuss performance implications of separate compilation of MATLAB functions.

Program constants, the second source of information, allow us to infer and propagate intrinsic type, shape, and size. MATLAB operators, the third source of information, allow us to extract type information in the case of operators that produce logical values, and shape and size information by considering the conformability requirements imposed by the operators.

Finally, MATLAB built-in functions, the fourth source of information, provide inference information for their output parameters based on the type of the input

[3]Since all variables in MATLAB are double precision, `real` and `complex` variables are declared in Fortran 90 as `double precision` and `complex*16` respectively.
[4]Our translator recognizes MATLAB version 4.2c; hence we do not consider the hierarchical matrices that were added to MATLAB 5.0.
[5]The use of this initial shape information can be turned off by a compiler flag.

parameters. This information is stored by the compiler in a database, referred to here as the *built-in result table*.

FALCON's type inference algorithms do not follow the traditional forward/backward strategy [Aho et al. 1985]. Instead, we use repetitive forward propagation through the SSA until a fixed point is reached. Backward inference seems to provide very little help for type inference in MATLAB programs because operators in MATLAB are overloaded and because all variables can change type during runtime. Consider, for example, the expression: `C = A + B`. If `A` is known to be an $n \times m$ `matrix`, `C` can be inferred to be a `matrix` of the same size. However, we cannot infer the shape or size of `B`, since it can be a `matrix` of the same size or a `scalar`.

The use of backward inference for intrinsic type also is hindered by possible type ambiguities in MATLAB. In general, MATLAB does not require any variable to be of a particular intrinsic type for an operation to be valid. For example, MATLAB permits logical operations on `complex` operands and accepts as array indices `real` and `complex` values, in addition to `integer` values. In the case of indexed assignments, however, we use a backward step to promote intrinsic types, as described in Section 4.2.

In this section we present FALCON's static inference mechanism. We first describe the algorithms for the intrinsic type inference. Next we describe the value-propagation technique used to improve the intrinsic type inference. Finally, we discuss the size and shape inference mechanism.

4.1 Intrinsic Type Inference

Although MATLAB operates only on `real` and `complex` values, FALCON's static inference mechanism also considers `integer` and `logical` values. The intrinsic types are inferred according to the following type hierarchy:

$$\texttt{logical} \prec \texttt{integer} \prec \texttt{real} \prec \texttt{complex}.$$

This hierarchy means that it is correct for variables with `integer` values to be declared as `real` or `complex` and for variables with `real` values to be declared as `complex`. In fact, intrinsic type inference could be avoided if all variables were considered `complex`. This, however, would affect the performance of the code due to the number of additional arithmetic operations required by `complex` variables.

The static mechanism for intrinsic type inference propagates intrinsic types through expressions using a *type algebra* similar to that described in Schwartz [1975] for SETL. For the case of logical operators, the result is always considered to be of intrinsic type `logical`. For the other operators and built-in functions, this algebra operates on the intrinsic type of MATLAB objects and is implemented using tables for all operations. For each operation, these tables contain the intrinsic type of the result as a function of the intrinsic type of the operands. When the intrinsic types of the operands are different, the static type inference defines the output of the expression to be of an intrinsic type that subsumes the intrinsic types of both operands according to the intrinsic type hierarchy described above. In some cases, such as division, the output is promoted to a higher intrinsic type from the type hierarchy, even if the intrinsic type of the operators are the same. For example, the division of two `integer` variables results in an output of intrinsic type `real`.

In some cases, the outcome of an expression or a built-in function can be of different intrinsic types, depending on the values of the operands (such as the power operator, square root, and inverse trigonometric functions). For example, for a `real` variable `A`, the outcome of $\sqrt{1-A}$ would be `real` if $A \leq 1$, and `complex` otherwise. We refer to these expressions as *ambiguously typed expressions*. In these cases, to improve the accuracy of the intrinsic type inference, we developed a value-propagation approach, described in Section 4.3, to infer statically the range of values of the variables. If this range of values is not sufficient to determine the intrinsic type, we promote the output type of the expression to `complex`. The general approach for intrinsic type inference is described in Algorithm 2.

Algorithm 2. *Intrinsic Type Inference:*

Input: An SSA representation of a MATLAB code segment.
Output: The intrinsic type for all variables in the program segment.

The algorithm is divided in two phases. The first phase traverses the program, setting the initial types according to the statement being visited. The second phase performs repetitive forward propagation through the program until a fixed point is reached. In this algorithm, **USES**(A_i) is the set of statements that uses A_i, and $\sqcap$ is the meet operator in the lattice shown in Figure 4.

Phase 1:

FOR each statement $\mathbf{S}_j$ in the program *DO*
CASE ($\mathbf{S}_j$)
Input (i.e., load(A_1, A_2, ..., A_n)):
for i=1 to n, A_i_Type ⇐ intrinsic type of the loaded variable according to the information from the loaded file.
Assignment from an expression or constant (i.e., A = <expr>):
A_Type ⇐ type algebra for <expr>.
Assignment from a built-in function (i.e., [A_1, A_2, ..., A_n] = F):
for i=1 to n, A_i_Type ⇐ intrinsic type according to the built-in result table.
Assignment from an α function:
Execute Algorithm 3.
Assignment from a λ function (i.e., A_i = λ(A_x,A_y)):
A_i_Type ⇐ A_x_Type ⊓ A_y_Type.
Assignment from a β function (i.e., A_i = β(A_p,A_f)), where A_p_Type is the intrinsic type of the previous definition:
(1) A_i_Type ⇐ A_p_Type.
(2) *MARK* $\mathbf{S}_j$.
OTHERWISE do nothing.
END CASE
END FOR

Phase 2:

WHILE marked statements ≠ ∅ *do*
TRAVERSE the program, checking each marked statement $\mathbf{M}_k$.

FOR each output variable $\mathtt{A}_i$ in $\mathbf{M}_k$ *DO*
oldType $\Leftarrow$ $\mathtt{A}_i$_Type.
IF $\mathbf{M}_k$ is an assignment from a β function *THEN*
$\mathtt{A}_i$_Type $\Leftarrow$ A_p_Type $\sqcap$ A_f_Type.
ELSE
UPDATE $\mathtt{A}_i$_Type according to *CASE* in Phase 1.
END IF
IF oldType $\neq$ $\mathtt{A}_i$_Type *THEN*
FOR each $\mathbf{S}_j$ in **USES**($\mathtt{A}_i$) *DO*
MARK $\mathbf{S}j$.
END FOR
ELSE
UNMARK $\mathbf{M}_k$.
END IF
END FOR
END TRAVERSE
END WHILE

Note that because `null` is an artificial type, in Figure 4 the combination of `null` with any other intrinsic type results in the non-`null` type. Notice also that 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, the dynamic test discussed below will guarantee that `complex` operations will be executed only when the variable has a `complex` value.

The algorithm for intrinsic type inference requires an iterative process due to the β functions and the fact that the forward definition has yet to be analyzed when it is encountered for the first time.

THEOREM 1. *Algorithm 2 terminates.*

PROOF. The iterative process is guaranteed to finish because, as shown in Figure 4, the inference system for λ and β functions uses a type lattice that is finite. Hence, in the worst case, the output of the β function and all variables that directly or indirectly use this output will be considered `unknown`, and the algorithm will terminate. □

THEOREM 2. *Algorithm 2 requires at most $O(nl)$ steps to terminate, where l is the maximum number of levels in the type lattice and n is the number of β functions in the program.*

PROOF. After a variable is placed in a lattice cell, it can never go back to a lower lattice level. Hence, in the worst case, in each iteration of the algorithm a single variable will advance one level in the lattice due to a β function. Thus, in the worst case, after $n \times l$ iterations all variables will be in the `unknown`, cell and the algorithm will terminate. □

In practice, however, since intrinsic type changes occur infrequently in well-written MATLAB programs, the convergence of Algorithm 2 takes place almost immediately. For example, in all 12 programs used to evaluate our translator, no variable was assigned the `unknown` intrinsic type.

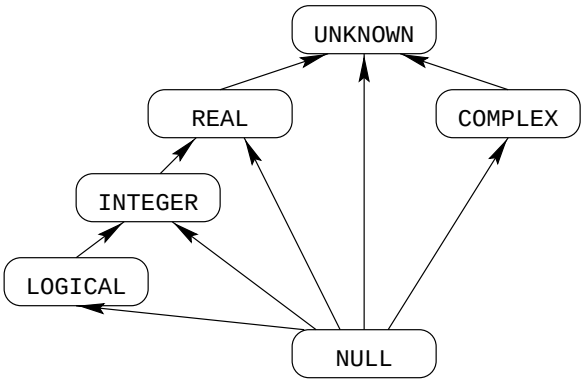

Fig. 4. Intrinsic type lattice for λ and β functions.

4.2 Intrinsic Type Inference of α Functions

We extend our forward propagation approach with a backward step to promote the intrinsic type of α functions whenever the intrinsic type of the RHS differs from that of the previous definition of the array. This backward step is performed on-demand, and the resulting α function type is promoted to one that subsumes both types. We observe that α functions occur normally inside loops, in which case the required tests and copy of the array generate an excessive overhead. Thus, if an α function output is inferred to be of a more general type than the type of the parameter array, the intrinsic type of the previous full definition of the variable is set to this more general type. Hence, as described in Algorithm 3, the backward step is necessary to update the previous full assignment to the variable and the subsequent indexed assignments. This on-demand backward approach was chosen over a full backward pass due to the small number of intrinsic types in our type hierarchy, which leads to (1) a fast convergence in the case of intrinsic type promotion and (2) the characteristics of the MATLAB language, where, as discussed in Appendix A, a variable can change type completely in different segments of the program.

Algorithm 3. *Intrinsic Type Inference of α Functions:*

Input: A code segment of the form

$$\begin{array}{ll} S_{full}\text{:} & A_{full} = ... \\ S_p\text{:} & A_p = ... \\ S_i\text{:} & A_i = \alpha(A_p, \mathrm{I}, \mathrm{RHS}) \end{array}$$

where A_{full} is the last full definition of A. If such a definition does not exist in the original program, an artificial definition with type `null` is created during the construction of the AST.
Output: The intrinsic type of the variable A_i.

Notice that in this algorithm an assignment from an α function is not considered a full definition of A. Hence, A_p may be different than A_{full}. Also, if the α function occurs inside a loop, A_{full} may be the β function corresponding to A.

Table I. Resulting Intrinsic Type for α Functions

Type of RHS	Type of previous definition					
	`Null`	`logical`	`integer`	`real`	`complex`	`unknown`
`null`	`null`	`logical`	`integer`	`real`	`complex`	`unknown`
`logical`	`logical`	`logical`	`integer`	`real`	`complex`	`unknown`
`integer`	`integer`	`integer`	`integer`	`real`	`complex`	`unknown`
`real`	`real`	`real`	`real`	`real`	`complex`	`unknown`
`complex`	`complex`	`complex`	`complex`	`complex`	`complex`	`complex`
`unknown`	`unknown`	`unknown`	`unknown`	`unknown`	`complex`	`unknown`

(1) *Compute* A_i__Type according to Table I
(2) *IF* (A_i__Type $\neq$ A_p__Type) *THEN*
(3) *IF* assignment to A_p is of the form $A_p = \alpha(...)$ *THEN*
(4) *FOR ALL* (A_k) assignments to A from S_{full} to S_p `DO`
(5) A_k__Type $\Leftarrow$ A_i__Type
(6) *END FOR*
(7) *ELSE*
(8) A_p__Type $\Leftarrow$ A_i__Type
(9) *END IF*
(10) *END IF*.

Notice that in this case we promote intrinsic types. Thus, in Table I, the confluence of any two known intrinsic types results in the highest intrinsic type, according to the type hierarchy, and the confluence of `unknown` and `complex` is `complex` rather than `unknown`, since `complex` is the highest intrinsic type in the hierarchy.

4.3 Value Propagation

The output of ambiguously typed expressions can be of multiple intrinsic types, not only due to the intrinsic type of the input parameters, but also due to the values of the input parameters during execution time. The simplest approach for these kinds of functions is to assume that the output of these built-ins will always be of type `complex`. This approach is followed by MathWorks [1995] in their compiler. However, this automatic promotion of intrinsic types may affect the performance of the code.

To improve the accuracy of intrinsic type inference in the presence of ambiguously typed expressions, FALCON applies value-propagation analysis. The goal of this analysis is to evaluate the range of possible values of each variable in the program in order to decide the intrinsic type of the outcome of ambiguously typed expressions. If the ambiguity is not resolved, then a promotion to `complex` occurs. FALCON's value-propagation analysis performs repetitive forward propagation through the SSA until a fixed point is reached. In each iteration the minimum and maximum possible values for each variable are estimated. Whenever it is not possible to infer any of these values for a particular expression, the values $-\infty$ or $+\infty$ are assigned as the minimum or maximum value respectively. For non-`scalar` variables, the estimated values correspond to the minimum and maximum possible values of all elements of the array. FALCON's value-propagation algorithm takes the following actions, depending on the kind of statement that is analyzed:

—Input (i.e., `load`): Assigns the range $-\infty$ to $+\infty$ to all loaded variables.

—Assignment statement with a constant: Assigns the constant as both the minimum and maximum values of the LHS.

—Assignment from an expression:

—If possible, evaluates the range of the expression as a function of the range of the operands, taking into consideration possible sign changes due to the expression (e.g., the range for x^2, where x has the range $-\infty$ to 1, will be 0 to $+\infty$).

—If not possible (e.g., when evaluating the range of $1/y$, where y has range -1 to 1), attributes the range $-\infty$ to $+\infty$ to the output variable.

—Assignment from a built-in function that returns a fixed range (e.g., `sin`, `cos`, `norm`, `rand`): If the input parameters are not `complex`, assigns the output range of the function.

—Assignment statement with a β or a λ function on the RHS: Assigns the minimum and maximum estimated values of the two parameters to the LHS. In the first iteration ignore the values from the forward definition of the β function.

—Assignment from an α function ($A_i = \alpha(A_p, \mathrm{I}, \mathrm{RHS})$): The range of A is

$$Range(A_i) = \min\,(Range(A_p), Range(\mathrm{RHS})) \text{ to } \max\,(Range(A_p), Range(\mathrm{RHS})).$$

If the range of estimated values of the RHS subsumes the range of estimated values of the previous definition of the variable, it updates its range and executes an algorithm similar to Algorithm 3 to update the previous full assignment to the variable.

4.4 Shape and Size Inference

Although MATLAB considers all variables to be two-dimensional arrays, the identification of `scalars` and `vectors` is important to avoid the unnecessary use of indices and the overdimensioning of variables. Unnecessary indices may increase computation time, and overdimensioning may lead to poor memory and cache utilization. Moreover, since MATLAB overloads operators with different semantics depending on the shape of the operands, the ability to recognize `scalars` and to differentiate between `rowVectors` and `columnVectors` is necessary. Consider, for example, the multiplication of two variables `A * B`. In MATLAB, depending on the shape of `A` and `B`, this can be an innerproduct, an outerproduct, a matrix multiplication, a matrix-vector multiplication, or a scalar multiplication (by another scalar, vector, or matrix). Due to performance and conformability issues, each of these possibilities requires a different method for the Fortran computation. Therefore, a different library function or operator is used for each case.

For each variable, the outcome of the static shape inference mechanism is an *exact shape* (i.e., `matrix`, `rowVector`, `columnVector`, or `scalar`) when all dimensions are known. Otherwise, if only one dimension is known, the variable is inferred to be `notMatrix` or `notScalar`; or, if no dimensions are known, the variable is inferred to have `unknown` shape.

Size inference also is important for the efficiency of the generated code. If the dimensions of the arrays are known during compile time, they can be declared statically, and the overhead of dynamic allocation can be avoided. However, in most cases for real programs it is impossible to statically determine array sizes

$$C = A + B$$
$$(c_1, c_2) = (a_1, a_2) \sqcap (b_1, b_2)$$

$a_1 = 1$ and $a_2 = 1$	$\Rightarrow$	$c_1 \leftarrow b_1$ and $c_2 \leftarrow b_2$
$b_1 = 1$ and $b_2 = 1$	$\Rightarrow$	$c_1 \leftarrow a_1$ and $c_2 \leftarrow a_2$
`for` $i = 1 : 2$,		
$a_i =?$	$\Rightarrow$	$c_i \leftarrow b_i$
$b_i =?$	$\Rightarrow$	$c_i \leftarrow a_i$
$a_i = b_i$	$\Rightarrow$	$c_i \leftarrow a_i$
`otherwise`,		error

Fig. 5. Rules for conformable operators.

Table II. Rules for the Exact Shape and Size for the Multiplication Operator

A ∗ B	B			
A	`scalar`	`vector`(b_1,1)	`vector`(1,b_2)	`matrix`(b_1,b_2)
`scalar`	`scalar`	`vector`(b_1,1)	`vector`(1,b_2)	`matrix`(b_1,b_2)
`vector`(a_1,1)	`vector`(a_1,1)	error	`matrix`(a_1,b_2)	error
`vector`(1,a_2)	`vector`(1,a_2)	`scalar`[1]	error	`vector`(1,b_2)[1]
`matrix`(a_1,a_2)	`matrix`(a_1,a_2)	`vector`(a_1,1)[1]	error	`matrix`(a_1,b_2)[1]

[1]Only if $a_2 = b_1$; otherwise error.

because they depend on the input. The only alternative in such cases is to allocate the array dynamically. An extra effort is necessary to predict the maximum size that a dynamically allocated array will take in order to avoid the overhead of multiple execution time tests and reallocations of a variable with every change in size. To this end, we use a symbolic-propagation technique, discussed in Section 5.2.1.

Shape and size information are obtained via *conformability analysis*. Similar analysis techniques, developed for size inference, are presented in Chatterjee [1993]. We define the operators that require conformability for only one of the dimensions of the operands (e.g., "∗", "/", and "\") as a *single-dimension conformable operator*, and operators that require both operands to have the same size (e.g., `+`, `-`, or logical operators) as a *size conformable operator*. The conformability analysis for a particular operator is done by applying the operator's meet (⊓) rules illustrated in Figure 5 for the + operator and, in Table II, for the ∗ operator. In these examples, the size information is indicated with the constants a_1, a_2, b_1, b_2, c_1, and c_2 that represent the exact values for the number of rows or the number of columns of the variables A, B, and C. `ColumnVectors` and `rowVectors` are represented as `vector`(x_1,1) and `vector`(1,x_2) respectively, and `unknown` values are represented by "?".

The algorithms for propagation of shape and size information through λ and β functions are very similar to Algorithm 2 for type inference, described in Section 4.1. During the static phase of the analysis, size information is obtained only from constants; hence, the lattice for size inference, presented in Figure 6(a), is similar to the one described by Wegman and Zadeck [1991] for constant propagation. For each dimension of a variable, there are only three possibilities for the lattice element: the bottom `null`, the top `unknown`, and all constant elements (κ) in the middle. There is an infinite number of κ_i lattice elements, each corresponding to a different value for the dimension; however, they are all in the same level in the lattice. The meet operator for λ or β functions is defined according to Figure 6(b).

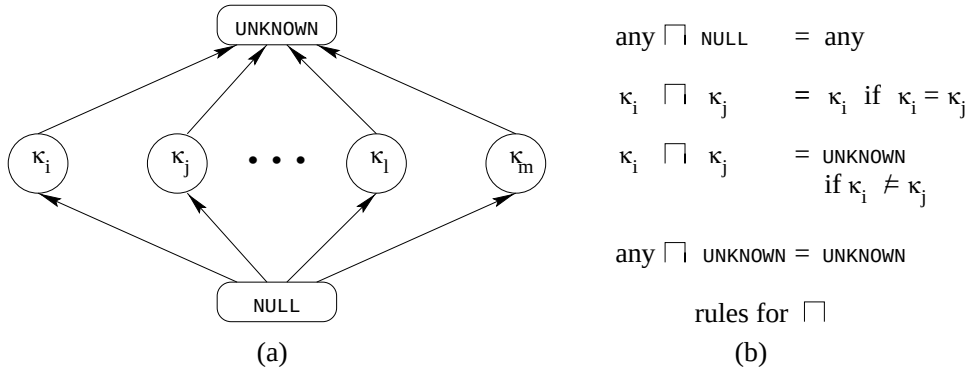

Fig. 6. (a) Lattice for static size inference, and (b) rules for the meet operator.

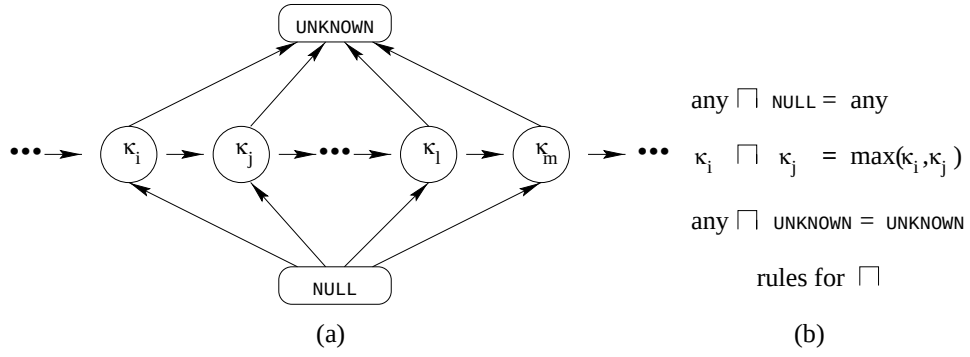

Fig. 7. (a) Lattice for static size inference of α functions, and (b) rules for the meet operator.

Due to the possibility of dynamic growth of arrays, the analysis of α functions for the static size inference requires a different approach. We use the lattice presented in Figure 7(a) for the analysis of α functions. This lattice is similar to the lattice in 6, but the constants κ are now ordered according to their values. The meet operator for α functions is defined according to Figure 7(b).

5. DYNAMIC PHASE

If any of the variable attributes necessary for the generation of the Fortran 90 declarations (i.e., intrinsic type, number of rows, and number of columns) have an `unknown` value at the end of the static inference phase, dynamic code to determine the necessary attribute at run-time must be generated. FALCON associates tags for each variable that has any `unknown` attribute value. These tags are updated at execution time. Based on these tags, which are stored in *shadow variables*, conditional statements are used to allocate the necessary space and to select the operation on variables of the appropriate intrinsic type. This section describes the overall strategy for the dynamic type inference and the dynamic size inference.

```
S1: temp = 1;
S2: X = function(temp);
    ...
S3: temp = 10;
S4: W = function(temp);
    ...
S5: temp = sqrt(-1);
S6: Z = function(temp);
    ...
S7: temp = 0.5;
S8  Y = function(temp);
```

(a)

```
INTEGER temp__1
COMPLEX*16 temp__2
DOUBLE PRECISION temp__3
...
temp__1 = 1
X = function(temp__1)
...
temp__1 = 10
W = function(temp__1)
...
temp__2 = sqrt(-1)
Z = function(temp__2)
...
temp__3 = 0.5
Y = function(temp__3)
```

(b)

Fig. 8. (a) MATLAB pseudocode example where the intrinsic type of the variable changes between assignments and (b) corresponding Fortran 90-generated code.

5.1 Dynamic Selection of Intrinsic Types

To avoid the excessive number of conditional tests necessary to detect the intrinsic type of the outcome of an expression, the dynamic inference mechanism considers only two intrinsic types: `real` and `complex`. If the intrinsic type of a variable can be determined statically, a Fortran declaration for the inferred intrinsic type is generated. Otherwise, the mechanism for dynamic definition of intrinsic types is activated. To this end, whenever a variable with `unknown` intrinsic type is used, a conditional statement is introduced during run-time to test the shadow value for intrinsic type. Each branch of the conditional statement receives a clone of the operation that uses the variable requiring the shadow test. In one branch, the variable is assumed to be of type `complex`, while in the other it is assumed to be of type `real`. In addition, two copies of the variable are declared, one `real` and one `complex`.

This solution introduces some overhead in the computation and increases the size of the code. However, in some cases it is cheaper than executing `complex` arithmetic all the time. One problem with this approach is that the number of possible type combinations grows exponentially with the number of operands. To reduce this problem, all expressions with more than two operands are transformed into a sequence of *triplets* using three-address code in the form: `t` $\leftarrow$ `x op y`.

The generation of dynamic code is avoided by using renaming whenever there is an assignment that redefines a variable such that its intrinsic type changes. Thus, for example, for the pseudocode presented in Figure 8(a), the compiler will declare three static instances of the variable, one for each intrinsic type assumed by the variable, and generate the code as shown in Figure 8(b).

Due to FALCON's static type inference mechanism, the generation of shadow tests for intrinsic type does not seem to occur very often in practice. In our test cases using actual MATLAB programs, there were no situations in which a shadow test for intrinsic type was necessary. Therefore, the question of whether this approach is better than just assigning `complex` intrinsic type to all variables requiring shadow tests was considered irrelevant.

```
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
```

Fig. 9. Example of shadow variables for size.

5.2 Dynamic Size Inference

Dynamic code is generated to compute during run-time the necessary space for dynamic allocation. To this end, two shadow variables are used to keep track of variable dimensions during execution time. For example, in an assignment of the form `A=B+0.5`, if the size of `B` is `unknown` after the static phase, the compiler would generate the code presented in Figure 9. This code uses shadow variables `A__D1`, `A__D2`, `B__D1`, and `B__D2` to store the run-time information about the size of each dimension of `A` and `B`. These shadow variables are initialized to zero at the beginning of the program. This figure shows the general form for the dynamic allocation. However, only the necessary statements are generated. If, for example, this were the first definition of variable `A`, statements `S1` and `S2`, which are needed only to reallocate the variable, would not be generated.

The use of shadow variables to keep track of array dimensions during execution time makes the generation of dynamic code straightforward for full array assignments. However, for indexed assignments, it is necessary to consider the possibility of dynamic growth of the array. Consider, for example, the simple indexed assignment: `P(k,k) = 4`, where `k` is an `integer scalar`. The dynamic code needs to check if any of the dimensions of the array `P` are going to be extended and, if so, reallocate `P` with its new size. Figure 10 presents the code generated for this indexed assignment.[6]

Some optimizations in the size inference mechanism are necessary to avoid the excessive number of tests and allocations. To illustrate this, consider the code segment of Figure 11(a), which computes the tridiagonal part of a Poisson matrix. If 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 S4, S11, and S13. The allocation test corresponding to S4 is shown in Figure 10; the other allocation tests would be equally complex.

The allocation tests before S4, S11, and S13 can be avoided if an allocation of `P` with size `n`×`n` was placed before S2. Avoiding the allocation tests is particularly important if there is no definition of `P` before S2, because if `P` were first referenced in S4, loop S2 would produce a very large overhead given that `P` would have to be reallocated at each iteration.

This simple example illustrates two static techniques needed to support dynamic size inference: *coverage analysis* and *efficient placement of dynamic allocation*. The objective of the first technique is to determine whether an indexed array assignment will not increase the size of the array. If this information is known at compile

[6]Another option would be to use pointers and perform only one allocation, but the copy of the old data into the new allocated memory space would still be necessary.

```
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
P(k , k) = 4
```

Fig. 10. Fortran 90 allocation test for the MATLAB expression `P(k,k)=4`.

```
S0:  load %(n)

S2:  for k=1:n

S4:    P(k,k) = 4;
S5:  end


S8:  for j=1:n-1;


S11:   P(j,j+1) = -1;

S13:   P(j+1,j) = -1;
S14: end
```

(a)

```
S0:  load %(n1)
S1:  P5 = null
S2:  for k1=1:n1
S3:        P6 = β(P1, P5);
S4:        P1 = α(P6,(k1,k1),4);
S5:  end
S6:  P4 = λ(P1, P5);
S7:  T__11 = n1-1;
S8:  for j1=1:T__11
S9:        P8 = β(P3, P4);
S10:       T__21 = j1+1;
S11:       P2 = α(P8,(j1,T__21),-1);
S12:       T__31 = j1+1;
S13:       P3 = α(P2,(T__31,j1),-1);
S14: end
```

(b)

Fig. 11. (a) MATLAB code segment for the generation of a Poisson matrix and (b) its corresponding SSA representation.

time, an allocation test for the indexed assignment is not necessary. Otherwise, the allocation test must be generated and followed by the second technique, which places the test where it will minimize the overhead.

5.2.1 *Symbolic Dimension Propagation for Coverage Analysis.* To determine whether there is definition coverage, we use a simplified version of a demand-driven symbolic analysis algorithm developed by Tu and Padua [1995]. 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. For example consider the SSA representation of the pseudocode that computes the tridiagonal part of a Poisson matrix, presented in Figure 11(b). Using symbolic analysis, it is possible to conclude that

(1) in S2, the maximum value of the variable $\mathtt{k}_1$ will be $\mathtt{n}_1$,
(2) S4 creates a matrix P with size $\mathtt{n}_1 \times \mathtt{n}_1$,
(3) in S8, the maximum value of $\mathtt{j}_1$ will be $\mathtt{T\_1}_1$, i.e., equal to $\mathtt{n}_1\mathtt{-1}$,
(4) $\mathtt{T\_2}_1$ and $\mathtt{T\_3}_1$ are equal to $\mathtt{j}_1\mathtt{+1}$ and $\leq \mathtt{n}_1$ (using simple symbolic algebra capabilities), and
(5) the size of P is not expanded in S11 or S13. Therefore, it is not necessary to generate dynamic allocation tests for these statements.

To solve the coverage problem for an indexed array assignment, we symbolically compare the size resulting from the indexed assignment with the size of the array before the assignment. If the array does not grow, the allocation test is unnecessary. If an indexed assignment has no previous definition, then it can be considered a full assignment and allocated as such.

5.2.2 *Placement of Dynamic Allocation.* Whenever necessary, Algorithm 4 is used for the placement of dynamic allocation.

Algorithm 4. *Placement of Dynamic Allocation:*

Input: An α function that requires dynamic allocation (S_i: $A_i = \alpha(A_p, \mathrm{I}, \mathrm{RHS})$.
Output: Location in the AST of the allocation operation for the matrix..

(1) *Find* the definition of the variables that appear in the dimension expression. The statement where it is defined is referred to here as S_{def} (in the case that two variables define the matrix dimensions, S_{def} is the latest definition).
(2) *Find* the first full definition of the array (referred to here as S_{full}) preceding the α function.
(3) *IF* there is no previous definition for the assigned variable (e.g., Statement S4 in Figure 11(a)), the `ALLOCATE` can be placed at any point between S_{def} and the assignment.
THEN insert the `ALLOCATE` in the statement following S_{def}.

(4) IF an allocation test is required after S_{full}, it can be placed after both S_{full} and the last use of S_{full} (referred here as S_{use}).[7]
THEN insert the allocation test in the outermost location between S_{use} and S_i.

As an example, consider again the code segment in Figure 11. The variable that defines the symbolic dimension for `P` in S4 is `n`. Its definition (S_{def}) is S0. The variable `P` has no previous full definition. Therefore, the `ALLOCATE` statement can be placed anywhere between S0 and S4.

If possible, the dynamic allocation should be placed outside the outermost loop. If a variable is assigned inside a loop, it will generate a β function in the SSA representation that, as mentioned in Section 3, will have one previous definition. This previous definition corresponds to S_{full}. Hence, the last use of S_{full} is guaranteed to be outside the loop. Thus, if S_{def} is outside the outermost loop, the `ALLOCATE` statement can be placed after S_{def}, but outside the outermost loop, thereby avoiding loop overhead.

6. EXPERIMENTAL RESULTS

This section presents the experiments performed to measure both the effectiveness of the MATLAB-to-Fortran 90 compiler described above and the effectiveness of the internal phases of the inference mechanism. The performance of compiled codes was compared with the times corresponding to the interpreted MATLAB execution, the C-MEX files compiled by the MathWorks compiler, and Fortran 90 versions of the same algorithms we wrote by hand.

We ran 12 MATLAB programs on a single processor of an SGI Power Challenge with the 90MHz MIPS R8000 Processor and 2GB of memory, running IRIX 6.1. To avoid large execution times, especially in MATLAB, the time required for the experiments was controlled by setting the problem size and the numerical resolution (in the case of iterative problems). A subset containing five of these programs was used for an evaluation of the effectiveness of the internal phases of the inference mechanism. Table III presents a brief description of the test programs, including the number of lines of the MATLAB code, the number of lines in the hand-written Fortran 90 program,[8] the number of lines in the Fortran 90 program generated by FALCON, and the elapsed time in seconds for the compilation on the SGI Power Challenge. Most of the benchmark programs were obtained from the software resident on the MathWorks archive[9] for MATLAB-based books. SOR, CG, and QMR are described in Barrett et al. [1993]; AQ, CN, Di, and FD are described in Mathews [1992]; and Ga, RK, and EC are described in Garcia [1994]. The other two programs, 3D and IC, were obtained from colleagues.

Table IV presents the best execution times in seconds for all programs in our test set, using the four strategies mentioned above: MATLAB interpretation, compilation using MCC and FALCON, and compilation of hand-written Fortran 90 codes.

[7] λ and β functions are not considered for use of S_{full} in this algorithm.
[8] These programs include the code required to read the input files using the MATLAB external interface.
[9] In general, MATLAB programs used as examples in books can be obtained from MathWorks, via the Web page at http://www.mathworks.com/books/index.shtml.

Table III. Test Programs

Test programs	Problem size	Lines MATLAB	Lines F90	Lines FALCON	Compile time(sec.)
Successive Overrelaxation method (SOR)	420 × 420*	29	188	640	0.32
Preconditioned Conjugate Gradient method (CG)	420 × 420*	36	196	553	0.32
Generation of a 3D-Surface (3D)	51×31×21	28	201	349	0.21
Quasi-Minimal Residual method (QMR)	420 × 420*	91	290	1139	0.38
Adaptive Quadrature Using Simpson's Rule (AQ)	1 Dim. (7)	87	192	581	0.33
Incomplete Cholesky Factorization (IC)	400 × 400	33	79	498	0.30
Galerkin method to solve the Poisson equation (Ga)	40 × 40	48	171	521	0.25
Crank-Nicholson solution to the heat equation (CN)	321 × 321	29	169	345	0.24
Two body problem using 4th order Runge-Kutta (RK)	3200 steps	66	168	455	0.30
Two body problem using Euler-Cromer method (EC)	6240 steps	26	139	298	0.21
Dirichlet solution to Laplace's equation (Di)	41 × 41	39	247	418	0.24
Finite Difference solution to the wave equation (FD)	451 × 451	28	155	306	0.21

* These programs used as input data a stiffness matrix from the Harwell-Boeing Test Set (BCSSTK06).

The Fortran 90 programs were compiled using the SGI IRIX 6.1 native Fortran 90 compiler with the optimization flag "O3." The C-MEX-files generated by the MathWorks MATLAB-to-C Compiler (MCC) were compiled both with the SGI IRIX 6.1 native C compiler using the optimization flag "O2" and with the GNU C compiler using the optimization flag "O3." For each program, the best execution time out of the two compilers was chosen. MCC does not support the `load` statement; hence, the input data were loaded using interpreted MATLAB commands (this `load` time was not timed) and provided to the MEX-files as function parameters. Assertions indicating the intrinsic type and shape of the loaded variables were added to the M-files to provide MCC with the same information that was extracted by FALCON's compiler from the loaded variables.

Figure 12 presents the speedups over the interpreted MATLAB execution that were obtained with the three sets of compiled codes: hand-coded Fortran 90, Fortran 90 programs generated by FALCON, and C MEX-files generated by MCC. Due to the large difference in performance for some of the programs, the speedups are shown in logarithmic scale.

The following sections discuss the performance of FALCON's compiler with respect to the interpreted MATLAB programs, the hand-written Fortran 90 programs, the performance of the C-MEX files generated by MathWorks MCC Compiler, and, finally, an evaluation of the inference phases of FALCON's compiler.

Table IV. Execution Times (in seconds) Running on an SGI Power Challenge

Program	MATLAB	MCC	FALCON	Hand coded
SOR	18.12	18.14	2.733	0.641
CG	5.34	5.51	0.588	0.543
3D	34.95	11.14	3.163	3.158
QMR	7.58	6.24	0.611	0.562
AQ	19.95	2.30	1.477	0.877
IC	32.28	1.35	0.245	0.052
Ga	31.44	0.56	0.156	0.154
CN	44.10	0.70	0.098	0.097
RK	20.60	5.77	0.038	0.025
EC	8.34	3.38	0.012	0.007
Di	44.17	1.50	0.052	0.050
FD	34.80	0.37	0.031	0.031

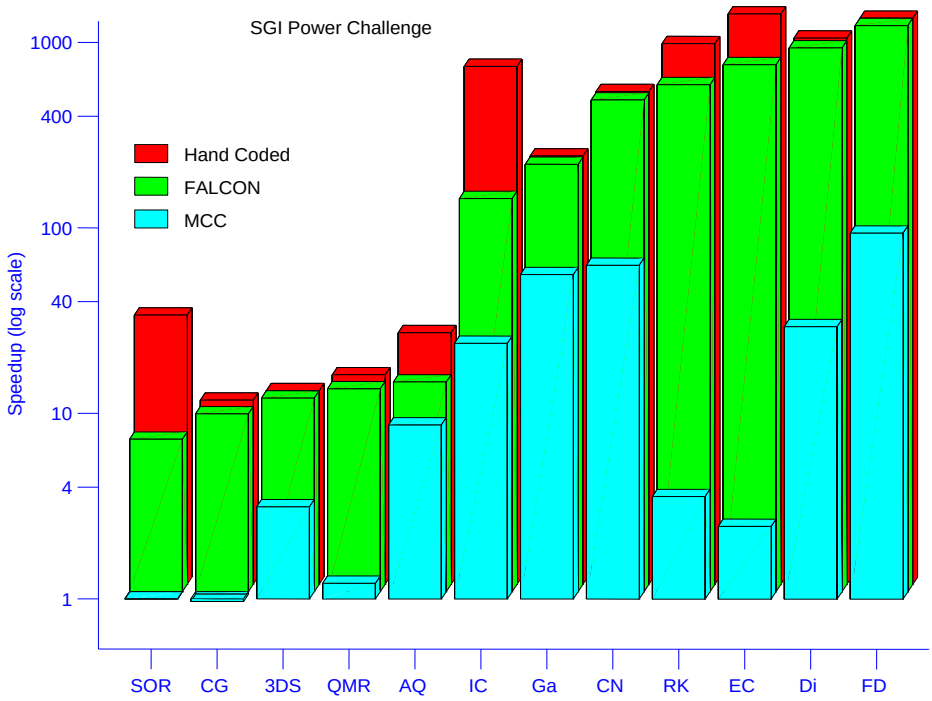

Fig. 12. Speedup of compiled programs over MATLAB.

6.1 Comparison of Compiled Fortran 90 Programs to MATLAB

Our experimental results show that for all programs in the test set, the performance of the compiled Fortran 90 code generated by FALCON is better than the respective interpreted execution, and the range of speedups is heavily dependent on the characteristics of the MATLAB program.

Programs that have execution time dominated by built-in functions (CG, SOR, QMR, and 3D) have a relatively small speedup (7 to 12) compared to MATLAB.

The reason, is that in general, the built-ins use the same optimized library functions that are called by FALCON's compiler and by the MathWorks compiler.

Programs that perform mostly elementary scalar operations using `scalars` or elementwise access of `vectors` or `matrices` (FD, Di, EC, RK, and CN) are the ones that benefit the most from compilation. This improvement is due to the more efficient loop control structure of the compiled code and to the larger overhead of the indexed assignments within the interpreted code. IC also performs elementary scalar operations; however, as discussed in Section 6.2, its FALCON speedup is hindered by an intrinsic type promotion.

Finally, the speedup obtained by AQ resulted from the better handling of indexed assignments by the compiled program. However, according to a scalability study to determine how problem size affects the relative speed of the programs [De Rose 1996], this improvement varies considerably according to 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.

The program AQ uses the Simpson's rule to numerically approximate the value of the definite integral:

$$\int_{-1}^{6} 13 * (x - x^2) e^{\frac{-3x}{2}} dx.$$

The adaptive quadrature method adjusts the integration interval to smaller subintervals when some portions of the curve have large functional variation [Mathews 1992]. This refinement in the integration process requires data movements and dynamic reallocation of an array as new subintervals are generated.

To measure the effect of these reallocations on the performance of the AQ program, we ran three sets of experiments varying the tolerance from 10^{-8} to 10^{-15}. The reduction in tolerance has a direct effect on the number of subintervals required by the algorithm. In the first set of runs, there were no preallocations[10] of the array. Thus, reallocation was necessary every time the algorithm required a refinement of the interval into subintervals. In the second set, there was an initial preallocation of 2500 subintervals. Hence, reallocation of the array was necessary only if the program required more than these 2500 subintervals. Finally, in the third set, there was an initial preallocation of 10000 subintervals, which was sufficient for all runs. Therefore, in this case, no reallocations of the array were necessary. Figure 13 presents the speedups of the compiled code over MATLAB running on the SGI Power Challenge. We notice that the speedup decreases as the memory operations (data movements and reallocations) of the programs start to dominate the execution time.

We observe that the speedup curves for the runs with full preallocation and the runs with 2500 intervals preallocated intersect around 2500 subintervals. The smaller speedup of the runs with full preallocation before the intersection is due to the overhead of allocating and using a larger array than necessary. We also observe that the speedup practically doubles when comparing the runs with full preallocation and the runs with no preallocation. The gap between these two speedup

[10] A common practice of MATLAB programmers is to preallocate `vectors` and `matrices` using built-ins, such as `zeros`, to avoid allocation overhead inside of loops.

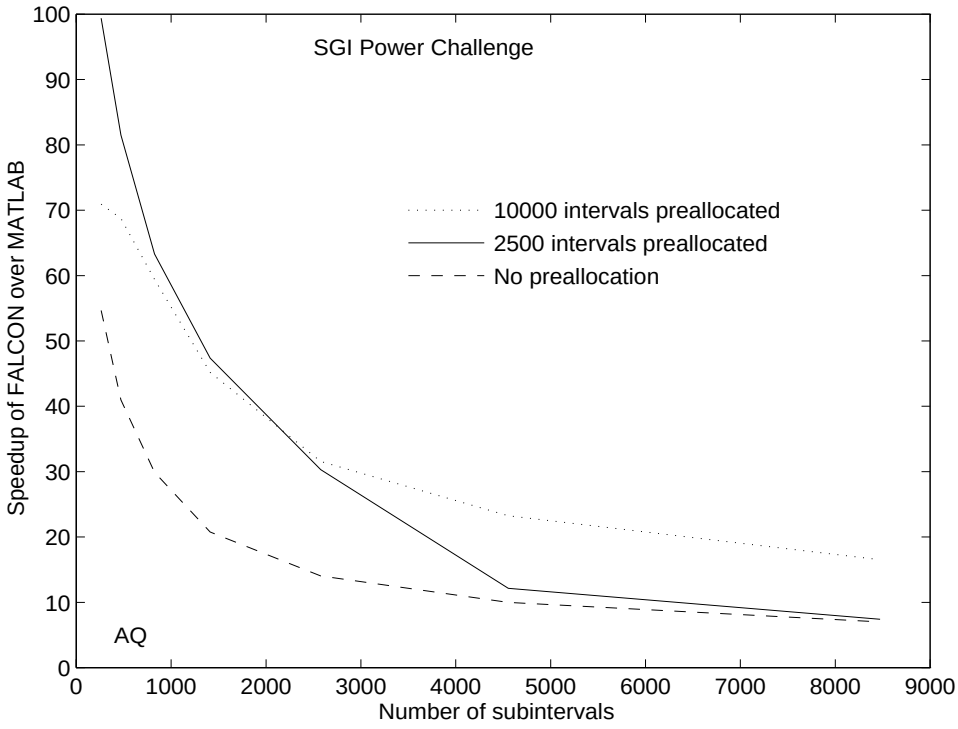

Fig. 13. AQ speedup when varying the required number of subintervals.

curves demonstrates the importance of inference techniques for preallocation of arrays, discussed in Section 5.2. In this program, however, the inference mechanism cannot avoid the overhead of reallocation by determining the correct size required by the array, because the number of required subintervals is dependent upon the input function and the tolerance.

6.2 Comparison of Compiler-Generated Programs with the Hand-Written Fortran 90 Programs

Figure 14 presents the speedups of the hand-written Fortran 90 programs over the compiler-generated versions. For most programs, the performance of the compiled versions and the hand-written programs is the same. The largest performance differences occur with IC and SOR. In both cases, the hand-written code performed more than four times faster than the compiler-generated code.

The reason for the performance difference with the IC program was the inability of the inference mechanism to detect that the conditional statement S3, shown in Figure 15, would prevent the array `L` from becoming `complex`. Due to the ambiguously typed expression (`sqrt`) in S2, the variable `r` is inferred to be `complex`. Thus, `L` in S7 also is inferred to be `complex`. However, since both `L` and `S` are `real`, the result of the square root function in S2 can be a `real` nonnegative value or a `complex` value with its real component equal to zero. Thus, due to the conditional

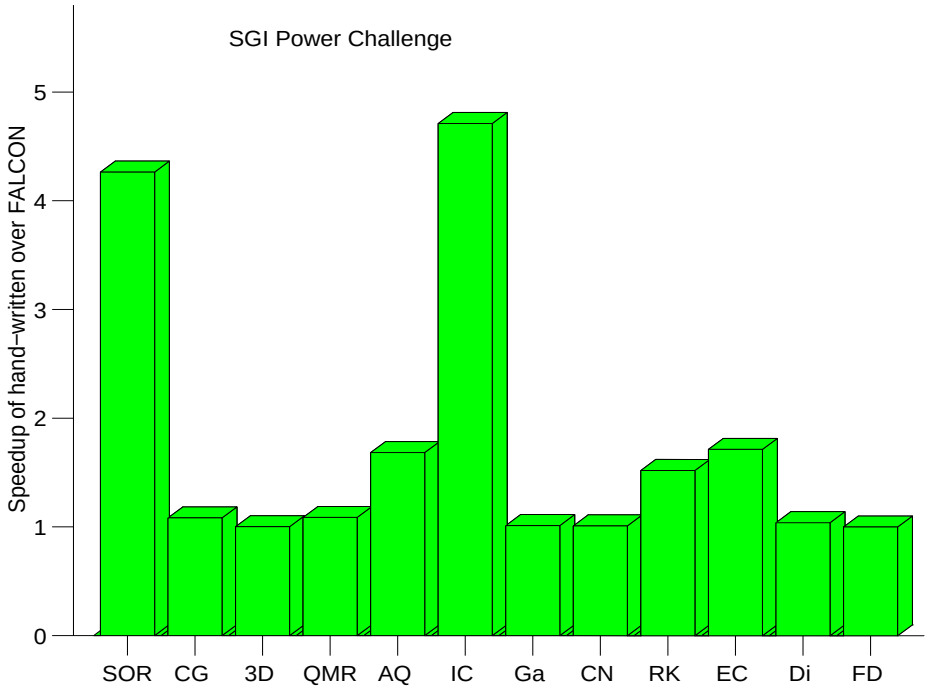

Fig. 14. Speedup of hand-coded Fortran 90 programs over the compiler-generated versions.

```
S0:  load %(L) (L is initially real)
S1:  for j = 1:n
     ...
S2:    r = sqrt(L(j,j) - S);
S3:    if (r <= 0)
S4:      Error = j;
S5:      L(j,j) = 1;
S6:    else
S7:      L(j,j) = r;
S8:    end
       ...
S9:  end
```

Fig. 15. MATLAB code segment for the Incomplete Cholesky Factorization (IC).

statement[11] S2, S7 will only be executed if `r` is `real`. Since the inference mechanism is unable to infer the `real` intrinsic type for the array `L`, the compiled code performs `complex` arithmetic for most of the program, whereas the optimized program uses `real` variables for the same operations.

The main reason for the performance degradation in the SOR case is the computation of the following expression inside a loop:

[11]MATLAB accepts `complex` variables in comparison operations, but takes into consideration only the real component of the variable.

```
x = M \ (N * x + b);
```

where `x` and `b` are vectors, `M` is a lower triangular matrix, and `N` is an upper triangular matrix. The hand-coded version considers the size of `M` and `N` and calls specialized routines from the BLAS library to compute the solve operation (\) and the matrix multiplication (`N * x`) for triangular matrices. The compiled version (as well as MATLAB) uses a run-time test to detect that `M` is a triangular matrix, and computes the solve using specialized functions. However, $\mathcal{O}(n^2)$ operations are needed to detect the triangular structure of the matrix. Moreover, in both cases, the matrix multiplication is performed using a generalized function for full matrices that performs $2n^2$ operations, whereas the BLAS function for triangular matrix multiplication performs roughly half the number of operations. Furthermore, it would not be worthwhile to test if the matrix is triangular during run-time because, as mentioned above, the test itself has an $\mathcal{O}(n^2)$ cost. A structural inference mechanism could be developed to detect matrix structures.

Other performance differences that are worth mentioning are those of EC, RK, and AQ. For the first two programs, the difference in performance is due to the built-in "`norm`." Both EC and RK compute several norms of vectors of two elements. The compiled programs call a library function for the computation of the norm, whereas the hand-written programs perform the computation using a single expression, which takes into consideration that the vector has only two elements.

Finally, the performance difference observed with AQ is primarily the result of the code generated by the compiler for the reallocation of matrices, as presented in Figure 10. In the hand-written code, because of 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.

6.3 Comparison with the MathWorks MATLAB Compiler

Figure 16 presents the speedups of the codes generated by FALCON over MCC's codes. We observe that all codes generated by FALCON ran faster than their MCC counterparts. In similar experiments, running on a SPARCstation 10 [De Rose 1996], we observed that in three cases (CG, SOR, and QMR) MCC generated programs that ran slower than MATLAB, whereas all codes generated by FALCON ran faster than MATLAB.

Three programs generated by FALCON (RK, EC, and Di) had significantly better performance than the corresponding MCC versions. The primary reason for these differences is the limitations of the inference analyses by the MathWorks compiler. The MathWorks compiler does not seem to perform use-coverage analysis and simplifies the handling of ambiguously typed expressions by assuming that they always return `complex` output. Moreover, as described by MathWorks [1995], the code generated by MCC cannot handle `complex` values nor perform subscript checking. To solve these problems, the code generated by MCC calls MATLAB using "*call-back functions*" provided in their library.

RK and EC perform several elementary vector operations using vectors of size 2. The code generated by MCC is very inefficient for these kinds of operations because it calls the MATLAB functions to perform `vector` and `matrices` operations. These call-back functions generate an overhead that, in this case, is not amortized due

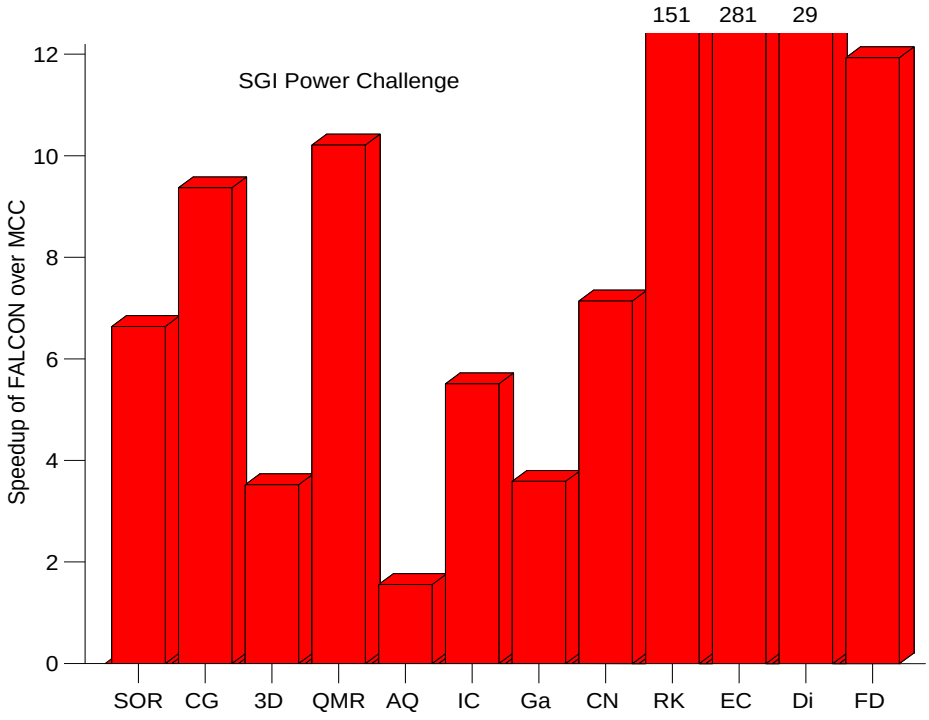

Fig. 16. Speedup of FALCON's compiler over MCC.

to the size of the vectors. Moreover, MCC does not infer the shape `vector` and, instead, treats both `vectors` and `matrices` with the same data structure. Furthermore, the lack of preallocation of variables in the MATLAB code also is responsible for the degradation of the performance of these MCC codes. FALCON's compiler, by contrast, is able to allocate all matrices in these programs outside the main loop, due to its symbolic propagation analysis.

Finally, the better performance from Di results from the value-propagation analysis. In this case, the intrinsic type inference mechanism can determine that the expression

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

will always return a `real` value between 1 and 2 for ω, whereas MCC assumes the output of this ambiguously typed expression to be of type `complex`. Although ω is only a scalar variable, for performance reasons it is important to be able to infer its intrinsic type; this variable is used to update an $n \times m$ rectangular grid which is in turn used to solve Laplace's equation in an iterative process. Therefore, if ω is assumed to be `complex`, the $n \times m$ matrix that contains the grid will have to be declared and operated as `complex`. Thus, the code generated by MCC for Di uses `complex` variables for most of its computations, whereas FALCON's generated code uses only `real` variables.

inference phases	AQ	CG	3D	Di	FD
no inference	27.93	3.49	3.92	2.45	2.93
only size and shape	1.82	2.46	3.40	0.28	0.34
no intrinsic type	1.80	2.42	3.40	0.26	0.12
only intrinsic type	27.85	1.68	3.58	2.58	2.07
no symbolic	1.48	0.59	3.16	0.09	0.13
all phases	1.48	0.59	3.16	0.05	0.03

Table V. Execution times in seconds when inference phases were deactivated.

6.4 Evaluation of the Inference Phases

For the analysis of the major phases of the compiler, we ran a subset of the programs above, with each program representing a different characteristic of the MATLAB codes. The following programs were selected:

—AQ, due to its reallocation of arrays;

—CG, representing the group of programs without indexed assignments;

—3D, due to its extensive use of library functions (`eig`);

—Di, due to its use of an ambiguously typed built-in; and

—FD, representing the elementary-operation-intensive programs.

With the use of compiler flags, we independently deactivated intrinsic type inference, size and shape inference, and symbolic dimension propagation. When type inference was deactivated, all variables, with the exception of loop indices and temporaries used for conditional statements, were declared `complex`. When size inference and shape inference were deactivated, all variables, with the same two exceptions just mentioned, were declared as two-dimensional allocatable arrays. For all runs where size inference and shape inference were deactivated, symbolic dimension propagation also was deactivated, since it is an optimization of size and shape inference.

In addition, we evaluated the independent compilation of functions by using the loaded variables of each program as function parameters.[12] In all cases, the independent compilation of a function was equivalent to executing the program with type inference completely deactivated. This occurs because the types of most variables in a function are, in general, directly or indirectly dependent on the function parameters. Thus, the lack of parameter type information will tend to get reflected in the whole function.

Figure 17 shows a graphical comparison of the execution times for all programs, using all six combinations. The execution times in seconds are presented in Table V.

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, intrinsic type inference has practically no effect on its performance. Its overall

[12]The MATLAB programs used in our benchmark were originally written as functions. They were converted into programs by using a MATLAB script that reads the function's parameters from an input file before calling the function.

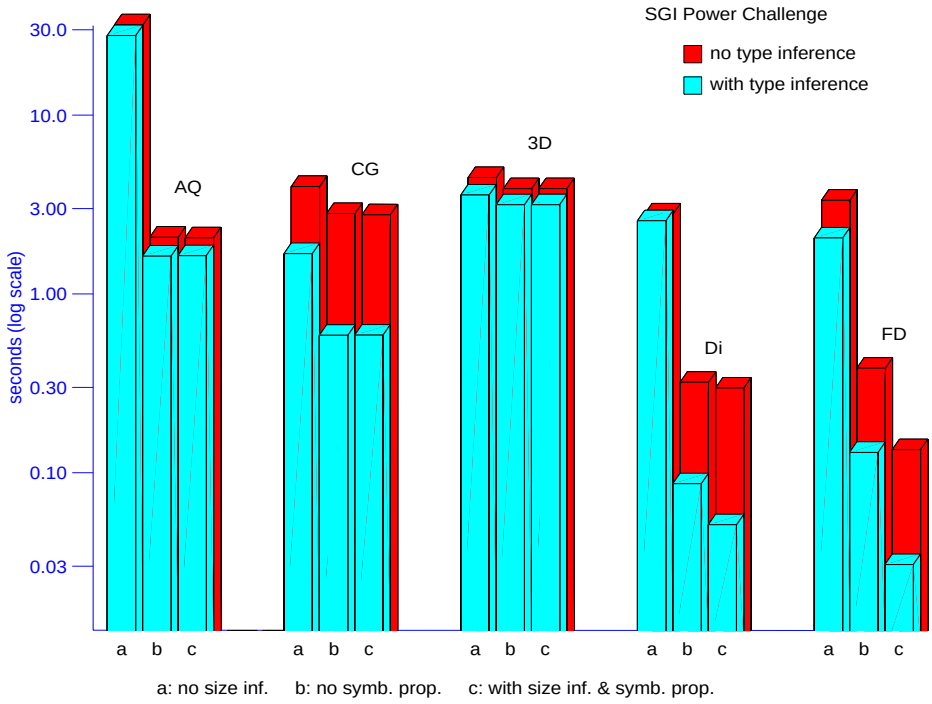

Fig. 17. Comparison of the inference phases.

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.

Size and shape 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 size inference, especially for scalars. When dynamic size 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 does not require enough work to compensate the overhead.

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

Intrinsic type inference generated the biggest improvements for computation-intensive programs (CG, Di, and FD). In these cases, when size inference and symbolic dimension propagation were activated, the speedup resulting from type

```
S0:  class Fruit {};
S1:  class Apple extends Fruit {};
S2:  class Banana extends Fruit {};

S3:  public class Example {
S4:    public static void main(String[] args) {

S5:      Fruit a = new Apple();
S6:      Fruit b = new Banana();

S7:      Apple x = (Apple)a;
S8:      Apple y = (Apple)b;
       }
     }
```

Fig. 18. Example of a type cast in Java that is correct at compile-time but generates a run-time exception.

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.

7. APPLICABILITY TO JAVA COMPILERS

Although the techniques described in this article were developed to MATLAB compilation, they also could be useful for other languages In this section we describe examples where our techniques would be beneficial for the compilation of strongly typed languages, such as Java.

Because of Java's strict semantics [Gosling et al. 1996], a native compiler for Java has to ensure safety by producing code that traps out-of-bound array accesses, null pointer references, and illegal type conversions. Due to these exceptions, it is generally difficult for a compiler to perform code optimizations [Moreira et al. 1998].

Consider for example the code segment shown in Figure 18, where both `a` and `b` are of class type `Fruit` and `Apple` and `Banana` are subclasses of `Fruit`.[13] This program should compile without problems, but `S8` will cause a run-time exception. To comply with the Java semantics, a native compiler should generate run-time type checks for all type casts in the program. However, if the compiler were able to prove that a particular type cast is always valid, the corresponding run-time type check could be avoided. Type inference techniques, such as the one described in Section 4.1, could be used by a native compiler to avoid the generation of unnecessary run-time type checks.

In Java, multidimensional arrays can be simulated only with arrays of arrays. There are several inconveniences with using these *Java arrays* in scientific computing. For example, storage is fragmented among rows, which requires both pointer chasing and indexing to access an element of a multidimensional array. Moreover, Java arrays are not regular structures, since rows can have different lengths. In Moreira et al. [1998], an "*array*" package that provide a collection of classes to implement true multidimensional rectangular arrays is proposed. The symbolic dimension propagation for coverage analysis, described in Section 5.2.1, can be easily

[13]If `S` and `T` are related classes (i.e., `S` and `T` are the same class, or `S` is a subclass of `T`, or `T` is a subclass of `S`) the casting conversion of a value of a compile-time reference type `S` to a compile-time reference type `T` is valid [Gosling et al. 1996].

Table VI. Inference Techniques for the Translation of MATLAB Programs

Technique	Section
Lexical scanner	
Differentiation between variables and functions	2
Static inference	
Intrinsic type inference of full array assignments	4.1
Intrinsic type inference of indexed array assignments	4.2
Conformability analysis for shape and size inference	4.4
Dynamic inference	
Shadow variables for intrinsic type and size inference	5
Dynamic allocation	
Symbolic dimension propagation for coverage analysis	5.2.1
Placement of dynamic allocation	5.2.2
Optimization	
Value propagation analysis for intrinsic type inference of ambiguously typed expressions	4.3

adapted in this context such that the Java compiler could prove that the array bounds of a code segment **S** are covered by the bounds of a previous segment **P**, and thereby, avoid the generation of out-of-bound traps in the code segment **S**.

Also, Moreira et al. [1998] proposed and implemented a *versioning* approach that divides code segments into safe and unsafe *variants*. The compiler produces a conditional test, which checks the extremes of the array bounds. If no out-of-bound array accesses are detected, the safe variant is executed. Otherwise, the unsafe variant is performed. Safe variants do not contain exception tests. Therefore, they can be optimized by the compiler. Unsafe variants, on the other hand, contain out-of-bound traps. The range propagation analysis technique, described in Section 4.3, could be used to prove that a code region is safe and, thus, avoid the conditional test and the generation of the unsafe variant.

8. CONCLUSIONS

The main goal of this work has been the development of inference techniques for the translation of MATLAB programs. These techniques are summarized in Table VI.

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, FALCON generates code that executes as fast as hand-written Fortran 90 programs, and more than 1000 times faster than the corresponding MATLAB execution on an SGI Power Challenge. Loop-based programs with intensive use of scalars and array elements (both vectors and matrices) are the ones that benefit the most from compilation.

When comparing the performance of the programs generated by FALCON's compiler with the performance of the programs generated by the MathWorks MATLAB-to-C Compiler (MCC), we observe that the performance of the programs generated by FALCON's compiler was faster than the corresponding programs generated by MCC, with the difference in performance ranging from approximately 3.5 to 280 times faster on the SGI Power Challenge. These differences in performance are attributed to the more enhanced inference mechanism utilized by FALCON's com-

piler.

Future enhancements under development include support for sparse computation, a structural inference mechanism, and the integration of the FALCON compiler with the Polaris restructurer [Blume et al. 1996]. The main goal of the structural inference is to help the code generator to select specific methods to perform certain operations, based on special matrix structures, such as `diagonal` and `triangular`. This structural information is propagated using an *algebra of structures* which defines how the different structures interact for the various operations. This information is then used by the code generator to replace general methods that operate on regular dense matrices with specialized functions for structured sparse matrices. Finally, the integration with the Polaris restructurer will allow us to study the effectiveness of parallelization techniques on MATLAB codes, in order to generate high-performance code for parallel architectures.

APPENDIX

A. AN OVERVIEW OF THE MATLAB LANGUAGE

This Appendix contains a brief description of MATLAB version 4.2c, which is recognized by the FALCON compiler. It is not our intent to cover MATLAB fully here. For a complete description, the reader is referred to the MATLAB User's Guide and Reference Guide, available from MathWorks, Inc.

MATLAB is an environment for numeric computation and visualization. It is an interactive system with its basic data element being a *rectangular matrix* that does not require dimensioning or intrinsic type declaration. MATLAB resembles conventional imperative languages, such as Fortran and C, in the form of its expressions and in its control structures, which include `for, if`, and `case` statements. Of today's most popular imperative languages, Fortran 90/95 is the closest to MATLAB because of its array syntax. In fact, both MATLAB and Fortran 95 make use of the triplet notation to specify array sections and contain an extensive number of intrinsic functions for array manipulation. However, the two languages are far from identical. For example, MATLAB does not contain `goto`, `forall`, or `where` statements, all of which are part of Fortran 95. There are also differences between the array manipulation functions of MATLAB and Fortran 95. In particular, MATLAB contains an extensive collection of functions to manipulate sparse matrices and reorder array elements, which are not part of Fortran 95. Also, MATLAB contains many functions implementing well-known linear algebra algorithms, which are not part of Fortran 95 either. Another significant difference is that Fortran 95 contains declarations and is usually compiled, whereas MATLAB is designed for interactive interpretation.

MATLAB is similar to APL [Pommier 1983] in that it can be executed interactively, is interpreted, and operates on aggregate data structures. However, the two languages have very different syntax. In fact, the only control structures in APL are unconditional and conditional branches. APL contains an extensive collection of array manipulation functions that enable the language to perform many operations without having to use iterative constructs. In fact, APL relies on expressions and function declarations to the extent that APL programs are often one-liners. Thus, in general, a MATLAB program tends to be larger than an APL program

for the same algorithm. Each APL intrinsic function is represented by a different symbol which, together with the reliance on expressions, gives APL programs an unconventional appearance. APL, developed in the early 1960s by K. Iverson, has been quite influential in the array language area, and several of its array operations, sometimes in simplified form, have been incorporated in Fortran 95 and MATLAB.

The rest of this Appendix describes MATLAB operators and language constructs, presents the three types of functions recognized by MATLAB, addresses how to create and access matrices, and describes the input/output options in MATLAB.

A.1 MATLAB Language Constructs and Operators

For this description it is important to differentiate a matrix from an *array*, which is the data structure used to represent matrices. Matrix operations are defined by the rules of linear algebra, whereas array operations are always performed element by element. MATLAB commands are intended to represent matrix computation and manipulation in the most natural form. In contrast, APL operators are primarily applied to arrays. For example, a matrix multiplication in APL is obtained by combining the operators "+", ".", and "×", whereas in MATLAB the operator "`*`" is overloaded to perform all linear algebra multiplications, such as inner product, outer product, matrix-vector multiplication, and matrix multiplication. For MATLAB operations such as `C = A + B` and `C = A * B` to be valid, MATLAB requires the operators to be conformable. Thus, if, for example, `A` is a 4×3 matrix and `B` is a 3×4 matrix, the multiplication operation would be valid, but the addition would result in a run-time error.

MATLAB is a structured language with assignable data structures, control flow statements, and an extensive set of functions. The control flow statements provided by MATLAB are `for` and `while` loops, conditional statements (`if`, `else`, and `elseif`), `return`, and `break`. These statements have the same semantics to the corresponding Fortran 90 counterparts, with the exception of the `for` loop. The general form of a `for` loop in MATLAB is

```
for v = expression
  statements
end
```

where the expression is actually a matrix. The columns of the matrix are assigned one by one to the variable `v`. In the kth iteration, the kth column of the matrix is assigned to `v`, and the statements are executed. When the expression is a triplet expression (*start:stride:end*), which in MATLAB corresponds to a row-vector, the `for` loop has the same semantics as the Fortran `DO` loop.

MATLAB expressions can be built using the usual arithmetic and logical operators and precedence rules (e.g., +, -, *, | for logical `or`, & for logical `and`, etc). MATLAB has two types of arithmetic operations: matrix operations, which, as mentioned above, are defined by the rules of linear algebra, and array operations, which are performed element-by-element. A decimal point (".") before the operator differentiates the array operations from the matrix operations (e.g., `A .* B` corresponds to the element by element multiplication of `A` and `B`). In addition, MATLAB provides a transpose operator and two division operators: "/" for *right division* and "\" for *left division*. The left division is convenient for the represen-

```
[nr,nc] = size(x);
avg = sum(x) / nc;
```

(a)

```
function avg = mean(x)
    [nr,nc] = size(x);
    avg = sum(x) / nc;
```

(b)

Fig. 19. M-file to compute the mean of a vector using (a) script and (b) function.

tation of the solution of linear equations (e.g., `x = A \ B` is the solution of the equation $Ax = B$). Notice that in MATLAB, $1/4$ and $4\backslash 1$ have the same numerical value.

In general, MATLAB displays the results of assignment statements and function calls. A semicolon at the end of the statement can be used to suppress the display of the output.

A.2 Functions

MATLAB has an extensive set of functions that provides most of its power. As opposed to APL, which limits its functions to zero, one, or two arguments, MATLAB allows functions with any number of input and output parameters. There are three types of functions in MATLAB: *intrinsic* or *built-in* functions, *MEX-files*, and *M-files*. Built-in functions range from elementary mathematical functions such as `sqrt, log`, and `sin` to more advanced matrix functions such as `inv` (for matrix inverse), `qr` (for orthogonal triangular decomposition), and `eig` (for eigenvalues and eigenvectors).

MEX-files are MATLAB-callable C or Fortran object codes that are built with a special interface module. They are called from MATLAB as if they were written in the MATLAB language. When the MATLAB interpreter reaches a call to a MEX-function, it recognizes the function as a compiled code, dynamically links it, and then performs the function invocation.

M-files consist of a sequence of normal MATLAB statements, which possibly include references to other M-files. There are two types of M-files: *scripts* and *functions*. A script does not accept parameters and operates on the scope of the caller. Hence, a script file is not *side-effect free*. Figure 19(a) shows a script file to compute the mean of vector `x`. Notice that the scope of the caller is used, and the variable `avg` is updated with the mean of `x`.

A function differs from a script in that arguments may be passed by value, and variables defined and manipulated inside the file are local to the function and do not operate on the workspace of its caller. Therefore, functions are side-effect free. Figure 19(b) shows a function to compute the mean of a vector. Notice, that, in this example, the body of the function is the same as the script file; however, the input variable is now given as an argument, and the output is defined in the function definition.

An identifier representing a function can become a variable if it appears in the right-hand side of an assignment. When used as a variable, the function becomes unavailable for the remainder of the program, or until the variable is *cleared*. However, in the case of loops, previous uses of the identifier are still considered to be a function. For example, Figure 20(a) presents a code segment in which the built-in "`i`" becomes a variable inside the loop. Figure 20(b) presents the corresponding

```
S1:  for k=1:3                          z = 1.0000 + 1.0000i
S2:    z = k + i                        x = 2
S3:    i = 1;                           z = 2.0000 + 1.0000i
S4:    x = k + i                        x = 3
S5:  end                                z = 3.0000 + 1.0000i
S6:  z = k + i                          x = 4
                                        z = 4
       (a)                                (b)
```

Fig. 20. (a) Example of a built-in becoming a variable inside a loop and (b) output from the execution of the code segment.

output from the code segment. We observe, that, in the body of the loop, the identifier `i` has ambiguous semantics. For all iterations of the loop, before `S3` the identifier `i` represents the intrinsic function `i`, which returns the basic imaginary unit $\sqrt{-1}$, whereas after `S3` the identifier `i` is a variable.

A.3 Creating Matrices and Accessing Matrix Elements

As opposed to Fortran 90, the MATLAB language has no statements to declare intrinsic type and dimensions of variables. As mentioned above, and in contrast to APL that supports multidimensional arrays, MATLAB works with only one type of object, a two-dimensional matrix,[14] which can be `real` or `complex`, and `full` or `sparse`. MATLAB recognizes the following forms of special matrices: scalars (1×1 matrices), row-vectors ($1 \times n$ matrices), column-vectors ($n \times 1$ matrices), and empty matrices (0×0 matrices). However, scalars, vectors, empty matrices, and text are all represented using this same structure. The intrinsic type of the matrices is determined and updated during run-time, and the storage is allocated dynamically.

There are several ways in which a matrix can be created in MATLAB. The most usual methods are

—Using functions that return matrices. For example, the function `rand(4,3)` returns a 4×3 matrix with uniformly distributed random entries.

—Entering an explicit list of elements. For example, the expression `A = [ 1, 2 3 ; 4, 5, 6 ]` creates a 2×3 matrix `A` with values 1, 2, and 3 in the first row and 4, 5, and 6 in the second row.

—Loading matrices from external files.

Notice that all matrices in MATLAB, with the exception of empty matrices, have lower dimensions set to 1. Matrix elements can be any MATLAB expression. For example, `B = [1, (1+2)*3; ones(1,2)]` will create a 2×2 matrix `B`, filled with 1 and 9 in the first row and ones in the second row. The storage is allocated automatically, up to the amount available on a particular computer. In addition, the matrix can be automatically increased to accommodate new elements. For example, considering the matrix `B` above, the expression `B(3,5) = 3` would result in the following matrix:

```
B =  1 9 0 0 0
```

[14]MATLAB version 5 allows the definition of multidimensional arrays. However, it does not provide functional support for matrix operations with these arrays.

```
S1:  A = rand(4,1)
     ...
S2:  A(5,5) = -1
     ...
S3:  A = 3i
```

Fig. 21. MATLAB pseudocode in which variable `A` changes shape and intrinsic type.

```
1 1 0 0 0
0 0 0 0 3
```

Notice that to insert the new element, the size of `B` is automatically increased to 3×5, and the undefined elements are set to zero.

In addition to size changes during run-time, MATLAB also allows intrinsic type changes during the program execution. This flexibility creates one of the main challenges for the compilation of MATLAB programs. Consider, for example, the code segment shown in Figure 21. The variable `A` will be a 4×1 column-vector `real`, filled with uniformly distributed random entries after `S1`, a 5×5 matrix `real` after `S2`, and a `complex` scalar after `S3`.

As shown above, matrix elements and submatrices can be referenced with indices inside parentheses. These indices can be expressed using constant values, expressions, and triplet notation (*start:stride:end*, where the stride is optional, and the use of a colon by itself in place of a subscript denotes the whole dimension). An expression used as a subscript is rounded to the nearest integer. For example, consider the matrix `B` above. The expression `B(sqrt(10),1:2:5)` would return the 1×3 row-vector [0, 0, 3]. A subscript can also be a vector. For example, if `x` and `y` are, respectively, $1 \times m$ and $1 \times n$ vectors, and the values of `y` are in the interval [0.5 ; m+0.5), `x(y)` will correspond to

```
[x(y(1)), x(y(2)), ..., x(y(n))].
```

If `x` and `y` are vectors, `A(x,y)` will result in the matrix obtained by taking the elements of `A` with row subscripts from `x` and column subscripts from `y`. This may be useful, for example, to reverse the columns of a matrix, which would be accomplished for the matrix `A` with the operation `A(:,n:-1:1)`.

The operator `[]` creates an empty matrix. With a few exceptions, all uses of an empty matrix propagates empty matrices. An empty matrix can be used to remove rows or columns of a matrix. For example, using the matrix `B` above, the expression `B([1,2], :) = []` will delete the first two rows of B, transforming it into a 1×5 row-vector.

`Complex` numbers are created with the use of the intrinsic functions `i` and `j`. Two possible ways to create a `complex` matrix `Z` are `Z = [1, 2; 3, 4] + i*[5, 6; 7, 8]` and `Z = [1+5j, 2+6j; 3+7j, 4+8j]`, which will produce the same result.

A.4 Input/Output

MATLAB provides both low-level and high-level commands for input/output. The MATLAB low-level I/O functions, which are not supported in the current version of our compiler, are based on the C language file I/O functions. It provides functions for file opening and closing (`fopen` and `fclose`), for unformatted I/O (`fread` and `fwrite`), for formatted I/O (`fscanf`, `fprintf`, `fgetl`, and `fgets`), for file position-

ing (`fseek`, `frewind`, `ftell`, and `ferror`), and for string conversion (`sprintf` and `sscanf`).

In addition, MATLAB provides the `load` and `save` commands, which are high-level constructs to retrieve and store variables on disk. These high-level constructs can import/export ASCII data files or double-precision binary files, which are known as *MAT-files*. MAT-files are created by the `save` command and read by the `load` command. When using the `save` command, the user can specify, using a variable list, that just a selected set of variables should be written. Otherwise, the whole workspace is saved. The `load` command, on the other hand, will always read the whole file; therefore, it does not take a variable list as a parameter.

ACKNOWLEDGMENTS

The authors would like to thank Kyle Gallivan, Stratis Gallopoulos, and Bret Marsolf for their suggestions throughout this work. We are particularly grateful to Stratis Gallopoulos for persuading us of the importance of MATLAB compilation several months before a current commercial MATLAB compiler was announced.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1985. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company.

Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. 1993. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM.

Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., and Tu, P. December 1996. Parallel Programming with Polaris. *IEEE Computer 29,* 12, 78–82.

Budd, T. 1988. *An APL Compiler*. Springer-Verlag.

Chatterjee, S. July 1993. Compiling Nested Data-Parallel Programs for Shared-Memory Multiprocessors. *ACM Transactions on Programming Language and Systems 15,* 3, 400–462.

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

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. October 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems 13,* 4, 451–490.

De Rose, L. A. 1996. Compiler Techniques for MATLAB Programs. Ph.D. thesis, University of Illinois at Urbana-Champaign, Department of Computer Science.

DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B., and Padua, D. 1995a. FALCON: A MATLAB Interactive Restructuring Compiler. In *Languages and Compilers for Parallel Computing*, C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 1033, Springer-Verlag, 269–288. 8th International Workshop, Columbus, Ohio.

DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B., and Padua, D. November 1995b. 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.

Gallopoulos, E., Houstis, E., and Rice, J. R. Summer 1994. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science & Engineering 1,* 2, 11–23.

Garcia, A. L. 1994. *Numerical Methods for Physics*. Prentice Hall.

Gilman, L. and Rose, A. 1984. *APL : An Interactive Approach*. Wiley.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison-Wesley.

Guibas, L. J. and Wyatt, D. K. 1978. Compilation and Delayed Evaluation in APL. In *Conference Record of the Fifth Annual ACM Symposim on Principles of Programming Languages*. 1–8.

Iverson, K. E. 1995. *J Introduction and Dictionary*. Iverson Software Inc. (ISI).

Mathews, J. H. 1992. *Numerical Methods for Mathematics, Science and Engineering*, 2nd ed. Prentice Hall.

MathWorks 1992. *MATLAB, High-Performance Numeric Computation and Visualization Software. User's Guide*. MathWorks.

MathWorks 1995. *MATLAB Compiler*. MathWorks.

Moreira, J. E., Midkiff, S. P., and Gupta, M. 1998. From flop to Megaflops: Java for technical computing. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98*. IBM Research Report 21166.

Moreira, J. E., Midkiff, S. P., Gupta, M., and Lawrence, R. D. 1998. Parallel data mining in Java. Tech. Rep. 21326, IBM Research Division.

Pommier, S. 1983. *An Introduction to APL*. Cambridge University Press, New York.

Schwartz, J. T. 1975. Automatic Data Structure Choice in a Language of a Very High Level. *Communications of the ACM 18*, 722–728.

Tu, P. and Padua, D. 1995. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing*. Barcelona, Spain, 414–423.

Wegman, M. N. and Zadeck, F. K. April 1991. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems 13,* 2, 181–210.

Received September 1997; revised May 1998; accepted July 1998