COMPILER TECHNIQUES FOR MATLAB PROGRAMS

BY

LUIZ ANTÔNIO DE ROSE

Bach., Universidade de Brasília, 1978
M.Stat., Universidade de Brasília, 1982
M.S., University of Illinois at Urbana-Champaign, 1992

THESIS

Submitted in partial fullfilment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

COMPILER TECHNIQUES FOR MATLAB PROGRAMS

Luiz Antônio De Rose, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1996
David Padua, Advisor

This thesis addresses the issues of translating an interactive array language, such as MATLAB[1], into a traditional compiled language, such as Fortran, in order to achieve better performance. It describes the main techniques, developed for our MATLAB-to-Fortran 90 compiler, to extract information from the high-level semantics of MATLAB for the generation of high-performance code. To perform the translation, an inference mechanism is necessary to generate the declarations for a *typed language*, to select the appropriate functions and operations, and to allocate the necessary space. The inference mechanism developed for this MATLAB-to-Fortran 90 compiler combines static and dynamic inference methods for intrinsic type, shape, rank, and structural inference. This inference is enhanced with value propagation and symbolic-dimension propagation analyses. The experimental results, which compare compiled generated programs with the corresponding interpreted MATLAB execution, show that the compiler can generate code that performs more than 1000 times faster than MATLAB on an SGI Power Challenge, and as fast as the corresponding hand-written Fortran 90 programs. When compared with the performance of C-MEX files generated by the MathWorks MATLAB compiler, we observed that, for our tests, the Fortran 90 programs ran faster than the corresponding C-MEX programs on an SGI Power Challenge and on a Sun SPARCstation 10. This better performance is mainly attributed to our enhanced inference mechanism.

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

To my wife, Jane

and my children: Pedro, Lígia, and Luiza.

Acknowledgments

I am indebted with the many people that helped me during all these years. First of all I would like to express my gratitude for my advisor, Professor David Padua. His support, guidance, assistance, and encouragement were a major factor during the development of this work. I am also grateful to Professors Gallivan and Gallopoulos, for their insights in many discussions about the FALCON project. Also, many thanks to the other members of my prelim and final committees, Professors Polychronopoulos, Saied, and Van Dooren; their suggestions about my research were very helpful.

I was very fortunate in having Bret Marsolf as my officemate for most of my years as a graduate student. I am not sure if I can express here how thankful I am for his friendship and help during all these years. I am also indebted to José Moreira for his friendship, encouragement, and all the technical discussions and suggestions.

I would like to thank Jairo, Moura, Luiz Antônio, and other friends from Brazil that encouraged and helped me to come to Illinois for my graduate studies. Many thanks also to Eduardo, Marielza, Creto, Vânia, Joe, Flor, Carlito, Vera, and all other friends from the Brazilian community here in Urbana-Champaign, for their support and encouragement on several occasions during all these years.

I wish to thank all the CSRD staff, especially Sheila for all the paper and thesis reviews, Donna for her secretarial support, and Bob, Wayne, and Gabriel, for their hardware and software support. Thanks to Mei-Qin Chen, Randall Bramley, and Jung Ah Lee, for providing MATLAB programs that were very helpful for debugging and evaluating the performance of the compiler, Bill Pottenger for his tips on how to use and measure performance on the

SGI, and Carlos Velez for writing the inliner and the untripletizer. I would also like to thank Alan Durham and Henry Neeman for all those long discussions in our qual study group, and all my other fellow graduate students and officemates for their friendship, help, and support during all my years as a graduate student.

I am very thankful to David and Ruth Krehbiel for all the family support that they provided us. This support was very important for me and my family. Also, thanks to all our friend from CERL and TCBC. They helped my wife and myself to feel at home here in Urbana-Champaign.

Thanks to my parents and to my wife's parents for their continuous support during all these years in graduate school, and most important, I would like to express my deepest gratitude to four special people in my life, my wife Jane, and my children Pedro, Lígia, and Luiza. My wife for understanding me and giving me encouragement and support, especially during the difficult times, and my children for adding joy and happiness to my life.

Thanks to the National Center for Supercomputing Applications for providing some of the computational environment. Initially my doctoral studies were supported by the Brazilian National Council of Research and Development (CNPq). This work was supported in part by the CSRD Affiliates under grant from the U.S. National Security Agency, and by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

Contents

Chapter

List of Tables

List of Figures

Chapter 1

INTRODUCTION

The development of software for scientific computation on high-performance computers is a very difficult and time-consuming task, requiring not only an understanding of the algorithms to be implemented, but also a detailed knowledge of the target machine and the software environment. Several approaches to facilitate the development and maintenance of programs for high-performance computers are currently under study. One approach is the automatic translation of conventional programming languages, notably Fortran, into parallel form. Polaris [BEF+94] and Parafrase-2 [PGH+89], developed at Illinois, and SUIF [AALL93], a compiler developed at Stanford, are examples of this first approach. Another approach is to extend conventional languages with simple annotations and parallel constructs. This second approach also involves the development of translation techniques for the extensions. Examples include High Performance Fortran [Hig93] and pC++ [BBG+93]. A third approach is to accept a very high-level description of the mathematical problem to be solved and automatically compute the solution in parallel. Examples of this approach are //ELLPACK [HRC+90], developed at Purdue, ALPAL [Coo88], developed at Lawrence Livermore Laboratories, and EXTENT [DGK+94], developed at Ohio State University.

We addressed the problem of development of software for scientific computation on high-performance computers by designing FALCON [DGG+95b, DGG+95a], a development environment that combines the three approaches described above.

1.1 High-Level Approach for Software Development for Scientific Computation

We believe that the development process should use a very high-level language that should be as close as possible to the mathematical description of the problem, albeit in the form of a simple and easy-to-use procedural language. The program written in this very high-level language is automatically (or semi-automatically) translated into a conventional programming language, such as Fortran, and, with the help of a parallelizer compiler, the program is finally translated into parallel form. The use of a very high-level language facilitates the development process by enhancing the ease of programming and portability of applications. Furthermore, the high-level semantics of the language can be exploited for the automatic generation of directives and assertions to help the parallelizer compiler.

FALCON includes capabilities for interactive and automatic transformations at both the operation-level and the function- or algorithmic-level. This environment supports the development of high-performance numerical programs and libraries by combining the transformation and analysis techniques used in restructuring compilers with the algebraic techniques used by developers to express and manipulate their algorithms in an intuitively useful manner [DGG+94]. The development process using FALCON starts with a simple prototype of the algorithm and then continues with a sequence of automatic and interactive transformations until an effective program or routine is obtained. The prototype and intermediate versions of the code are represented in an interactive array language.

1.2 Issues on the Utilization of Interactive Array Languages

Interactive array languages, such as APL [GR84, Pom83], and MATLAB [Mat92a] are powerful programming tools for the development of programs for numerical computation. Many computational scientists consider it easier to prototype algorithms and applications using

array languages instead of conventional languages such as Fortran and C. One reason is the interactive nature of the language, which facilitates debugging and analysis. A second reason is that interactive array languages are usually contained within problem-solving environments which include easy-to-use facilities for displaying results both graphically and in tabular form [GHR94]. Third, in these languages it is not necessary to specify the dimension, rank, or intrinsic type of elements of arrays. While some researchers may consider that lack of typing increases the probability of error, in practice programmers find that this is a convenient feature. Finally, these languages also have an extensive set of functions and higher-level operators, such as array and matrix addition, multiplication, division, matrix transpose, and vector reductions, that facilitate the development of scientific programs.

The downside is that interactive array languages are implemented with interpreters and, therefore, their execution is sometimes inefficient. The interpreter spends its time reading and parsing the program, determining the intrinsic type of the operations, dynamically allocating storage for the resulting variables, and performing the operations. In fact, a large fraction of the interpreter's execution time is wasted doing work that could be done statically by a compiler. For example, the compiler could determine the intrinsic type of elements and the dimensions and shape of many operands in the program by doing global flow analysis. In this way, the execution could be made more efficient by eliminating the need for some or all of the run-time bookkeeping operations. A study of the effectiveness of this type of approach on APL programs is presented in [Bud88]. When the bulk of the computations is done by the high-level array functions, the inefficiency of the interpreter is less of a problem. This is because these high-level functions are not interpreted and the bookkeeping operations need to be performed only when the function is invoked and/or returns. However, for some applications and algorithms, such functions are not sufficient and the program needs to execute a significant number of loops and scalar operations. In some experiments we have conducted [DPar], it was observed that interpreting programs executing mainly loops and scalar operations could be up to three orders of magnitude slower than

executing their compiled versions.

1.3 Compiling MATLAB

One important aspect of FALCON's design was the selection of its input language. In order to shorten the development process by having immediate access to existing support routines, and to take advantage of existing powerful graphics and other I/O facilities, we chose to use an existing array language as the source language for the FALCON system. MATLAB was chosen over other interactive array languages, such as APL, for several reasons, including its popularity and its simple syntax with exclusive use of structured constructs.

Fortran 90 is the natural choice for output language because of its many features that facilitate the compilation process, especially for vector computation. Examples of these constructions are array operations, such as addition and multiplication, that have the same semantics in Fortran 90, and some vector reductions and matrix operations, such as matrix multiplication and transposition, that can be directly translated into Fortran 90 functions. However, the translation process is not always straightforward; there are several idiosyncratic features in MATLAB that cannot be converted directly into Fortran 90. Examples of these features are the operators "/" and "\" that are used for division of matrices; the overload of operators, such as "*", that have different semantics depending on the rank of the variables being operated, and, most importantly, the lack of intrinsic type definitions and specification of dimensions of variables, and the possibility that any of these variable properties could change during run-time.

1.4 Problem Overview and Thesis Outline

This thesis addresses the issues of compiling an interactive array language. It describes the main techniques developed for the compilation of MATLAB programs. Some of the goals of this MATLAB compiler are: the efficient extraction of information from the high-level semantics of MATLAB; the generation of high-performance code for serial and parallel

architectures; and the use of the semantic information in order to facilitate the work of a parallelizing compiler.

As described previously, due to its interactive nature, its extensive set of functions and higher-level operators, and its lack of requirements for specification of intrinsic type and dimensions of variables, MATLAB is a very powerful programming tool for prototyping algorithms and applications. This lack of "declarations", however, is one of the major challenges for the translation. The fact that the language is not *typed* simplifies the programmer's job because every function and operator is polymorphic and, therefore, type casting or function cloning to accept different input and output types is not necessary.

Consider, for example, the simple computation of a square root function ($\sqrt{k}$). It may result in an INTEGER, REAL, or COMPLEX output, depending on the value of k. In MATLAB, however, the user does not need to be concerned about the intrinsic type of the input variable to compute this function; furthermore, the same statement works for any variable, independent of its intrinsic type, number of dimensions (rank), or size of each dimension (shape). Moreover, the intrinsic type, rank, and shape of the output variable are not important for the correct execution of the program. On the other hand, for a typed language such as Fortran and C, the user is required to know these properties for all the variables in order to select the correct method to compute the square root. Therefore, an inference mechanism is necessary to generate the declarations for a *typed language*.

The inference mechanism developed for our MATLAB compiler combines static and dynamic inference methods, and is enhanced with symbolic and value propagation analyses. The goal of our inference system is to determine in a MATLAB program the following *variable properties*:

intrinsic type: (e.g, COMPLEX or REAL);

rank: (e.g., VECTOR, MATRIX, SCALAR);

shape: (i.e., size of each dimension); and

structure: (e.g., UPPER TRIANGULAR, DIAGONAL, SQUARE MATRIX).

This thesis is organized as follows: related work is discussed in Chapter 2; an overall strategy is presented in Chapter 3; the internal representation of the compiler is presented in Chapter 4; the algorithms for the static inference mechanism are discussed in Chapter 5; the dynamic phase and its algorithms are discussed in Chapter 6; experimental results are presented in Chapter 7; and, finally, our conclusions are presented in Chapter 8.

Chapter 2

RELATED WORK

There are many examples of *typeless* programming languages, including APL, MATLAB, ML [GMW79], Haskell [HWA+88], Lisp [MAE+65], and Smalltalk [GR83]. These languages can be divided in two classes: the *statically typed* languages (e.g., ML and Haskell) and the *dynamically typed* languages (e.g., Lisp, Smalltalk, APL, and MATLAB).

Statically typed languages include type constraints as part of their language definition. Hence, type inference is necessary to ensure that these type constraints are satisfied. These type constraints allow compilers to deduce most of the type information, thereby making run-time type-checking unnecessary. For cases in which the type-checker cannot deduce the type of a valid expression, the user must provide the type information. Normally in these languages, the type inference must be accurate enough to satisfy the language's type constraints. Thus, the type-checker may sometimes find a more generic type assignment than expected.

A classical example is the language ML, a meta-language for theorem proving, where the main motivation for strict type-checking is to ensure that every computed value of type `theorem` is indeed a theorem [GMW79]. To facilitate type inferencing, several restrictions are imposed to the language. For example, ML does not allow expressions with mixed types. Thus, a function definition

```
fun successor(n) = n+1;
```

will be considered a function that takes an INTEGER as argument (`n`) and returns another

INTEGER. The reason for this is that the operator + can only be applied to two operands of the same type. Therefore, n must have intrinsic type INTEGER because it is being added to the INTEGER 1. Also, to allow the type to be inferred statically, ML sometimes requires explicit type information to resolve overloading. in these cases, the lack of this type information will result in run-time type errors.

Dynamically typed languages do not have type constraints; hence, run-time type-checking is a necessary part of the system. Due to the lack of type constraints, compilers cannot deduce all type information. Therefore, for the cases where the type-checker cannot deduce the type of a valid expression, run-time type checks are required. For dynamically typed languages, type inference is primarily used for optimization purposes. Hence, the inferred type information should be as precise as possible in order to avoid the overhead of run-time type-checking. We now first describe relevant type inference techniques, followed by some approaches for the compilation of APL, MATLAB, and MATLAB-like languages. These approaches range from research projects to commercial products.

2.1 Relevant Type Inference Approaches

Type inference algorithms have been presented in the literature for both classes of languages described above. However, most of the work concentrates on defining a type system and inference rules for a particular language. One of our main objectives in this thesis is to evaluate the effectiveness of the inference mechanisms for detecting intrinsic type, rank, and shape on MATLAB programs.

We make use in this work of data-flow analysis as described by Aho, Sethi, and Ullman in [ASU85], and type inference techniques developed for SETL [Sch75] and APL [Bud88, Chi86]. We extend these techniques where necessary to deal with peculiarities of the MATLAB language and to improve accuracy and performance. Examples of these techniques are: a structural inference mechanism, a symbolic dimension propagation analysis, and a value propagation analysis.

A forward/backward traversal scheme for type inference is described in [ASU85]. In this scheme, type inference is performed with data-flow analysis on a flow graph of the program. The *in* and *out* set of variables for each block of the program are mapped onto sets of possible types. The scheme uses an iterative process that propagates information forward and backward, reducing the set of types associated with each variable until a fixed point is reached. One assumption of this scheme is that variables do not change types during the execution of the program. In our case, this is not a valid assumption since variables in MATLAB can change types during run-time; thereby not always allowing a backward step. Hence, our inference mechanism concentrates on a forward data-flow analysis and, as discussed later, performs a backward step only when backward inference is possible.

SETL is a set-theoretically oriented language of very high-level. A SETL program may be considered to represent an algorithm before it is codified into a language of lower-level [Sch75]. It treats types in a fully dynamic way, with no type declarations. However, the types of the objects that appear in SETL programs can be deduced with the use of an appropriate global analysis [Sch75]. For this type inference, a *type algebra* that operates on the structural type of SETL objects is used. This algebra is implemented using tables whose entries describe the action, on the symbolic entities of the algebra, of each of the primitives of the language to be analyzed [Sch75].

2.2 Compilation of APL

APL is similar to MATLAB in that it can be executed interactively, is usually interpreted, and operates on aggregate data structures. A few compilers for APL have been developed in the past. These compilers are also based on forward/backward dataflow analysis.

Budd [Bud88] and Ching [Chi86] have independently developed compilers for APL. Budd's compiler translates APL programs into C, while Ching's compiler produces IBM System/370 assembly code directly. The motivation for their work was to investigate the issues raised by the development of a compiler for a very high-level language, and to exploit

the high-level semantics of the APL language. Hence, their work has several similarities with our project, but there are also some important issues that differentiate their work from ours. First, Budd's work was focused on demand driven evaluation to reduce operations and memory requirements, while our work concentrates on generation of high-performance parallelizable code. Second, both Budd's and Ching's procedure for undetermined types and shapes during compile time differs significantly from ours since arrays in MATLAB are often built using Fortran-like loops and assignments that may be distributed across several sections of the code. In contrast, the techniques developed for APL assume that arrays are usually built by a single high-level array operation. Also, APL doesn't have the type COMPLEX. Since operations with COMPLEX increase the processing time by a factor of two or more, the penalty for not inferring the type of one variable in our compiler is much larger than in an APL compiler. Finally, APL's syntax is much simpler than MATLAB. The syntax for APL expressions is so regular that it can almost be recognized by a finite state automaton [Bud88]. Additionally, all functions in APL, including user defined functions, are limited to either zero, one, or two arguments. This limitation facilitates the translation process. Therefore, due to the differences between the languages and the two approaches, as described above, the results of their research do not necessarily carry over to our research.

We make use of some of the techniques developed for these two languages (SETL and APL) and extend them, with techniques originally developed for Fortran, to analyze array accesses and to represent the gathered information in a compact form [TP95].

2.3 Compilation of MATLAB and MATLAB-like Languages

CONLAB [JKR92] is an interactive environment for developing algorithms for parallel computer architectures. It uses a subset of the MATLAB language, with extensions for expressing parallelism, synchronization, and communication. A translator from CONLAB to C was developed by Drakenberg et.al. [DJK93]. However, some simplifications and modifications

have been made to the source language to allow efficient C code to be produced, such as the exclusion of all primitives for synchronization and communication, except for message passing. Some sort of type inference system is alluded to by the authors in their papers, but it is not described.

A simple approach for the compilation of MATLAB was taken by [Ker95] to translate MATLAB into C++. In this work, a matrix class was created to take care of all type and shape inference decisions during run-time. This class is then utilized by the generated C++. So, effectively, the control structure is compiled, but all the mathematical operations are still interpreted within this matrix class.

Recently, MathWorks released MCC, a MATLAB Compiler [Mat95] that translates MATLAB programs into C for stand-alone external applications, or into C MEX-files which are called within the MATLAB environment. MEX-files are MATLAB-callable C or Fortran dynamically linked subroutines that are built with a special interface module. From [Mat95], it appears that MCC performs only simple inference and relies upon user provided flags, pragmas, and assertions[1] to optimize the generated C code. In Chapter 7 we compare the performance of the codes generated by MCC and by our compiler for a set of MATLAB programs.

[1]Assertions are M-files functions that are installed by the compiler and used to specify the intrinsic type or rank of a variable.

Chapter 3

OVERALL STRATEGY

3.1 Restrictions to the MATLAB Language

In our MATLAB compiler we tried to support as much as possible the current version of the MATLAB language (Version 4.2C). However, a few restrictions on the language were necessary, mostly to be able to perform the static inference.

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

- Currently, inter-procedural analysis within M-files is performed in our system by in-lining the M-files. Therefore, recursive M-files are not supported by this version of the compiler. In practice, recursive M-files do not occur very often due to the inefficiency of the interpreter in handling recursions.

- Functions that can receive as a parameter a string containing a MATLAB expression, or the name of an M-file to be executed (e.g., eval, feval) are not supported. Due to the possibility of changes in the value of the parameter during run-time, the compilation of these functions became practically impossible.

- Although not due to a limitation of our inference mechanism, the current version of our compiler does not support global variables or sparse constructions, and supports only a limited number of input/output commands. The support of global variables and the reminder of the input/output constructions is straightforward, and should be available in a second version of the compiler. Support for sparse computation requires more research. This issue is being addressed for the FALCON system in the work by Gallivan et. al. [GMBW95].

3.2 Phases of the MATLAB Compiler

The main challenge of the MATLAB compiler is to perform inference on the input program to determine the variable properties: intrinsic type, rank, shape, and structure. These properties are used by the compiler to generate the Fortran 90 declarations and to optimize the output code.

The MATLAB compiler was structured in a conventional way [ASU85] with a series of different passes, as shown in Figure 3.1. This section discusses the main issues and the overall strategy adopted for each of the phases of the compiler.

3.2.1 Structure of a MATLAB Program

MATLAB is a procedural language. Its current version works with essentially one kind of data structure: a rectangular numerical matrix [Mat92a]. A MATLAB program consists of one or more Fortran-like statements which may include function calls.

There are two types of functions in MATLAB: *intrinsic* or *built-in* functions, 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).

M-files consist of a sequence of MATLAB statements, which possibly include references to other M-files. There are two types of M-files: *scripts* and *functions*. A script does not

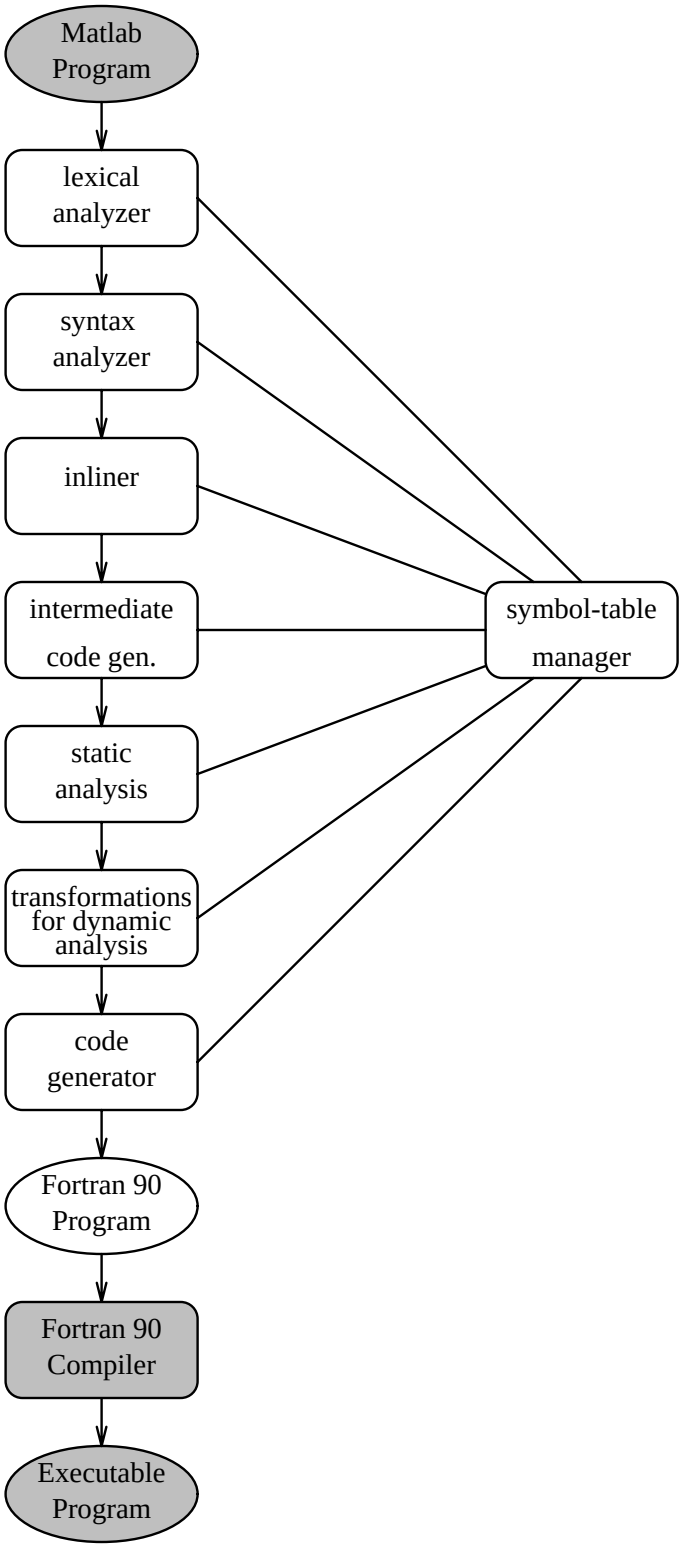

Figure 3.1: Phases of the MATLAB compiler.

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

(a)

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

(b)

Figure 3.2: M-file to compute the mean of a vector using (a) script and (b) function.

accept parameters and operates on the scope of the caller. Figure 3.2 (a) shows a script file to compute the mean of vector `x`. Notice that the variable `avg` is updated with the mean of `x`, that is a variable in the scope of the program of function that called the script.

A function differs from a script in that arguments may be passed, and that variables defined and manipulated inside the file are local to the function and do not operate on the workspace of its caller. Figure 3.2 (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 function's input variable is given as an argument, and the output is defined in the function definition. One important characteristic of M-file functions in MATLAB is that they are *side-effect free*. This functionality can be used to facilitate the exploitation of functional parallelism [GP92] and loop parallelism.

To avoid confusion, from here on when the term function is used alone in the text, it will refer to M-file functions and built-in functions only, and not to script files. Similarly, the term M-file will refer to M-file functions and not to script files.

3.2.2 Scanner, Parser, and the Symbol Table Generation

Initially, a *LALR(1) grammar* [ASU85] was defined for MATLAB, as there is no publicly available grammar for the MATLAB language. The compiler accepts as input a MATLAB script (from here on referred to as *main program*), that may contain several M-file calls. The lexical analyzer, which is generated by *lex* [LS], reads and converts the main program into a stream of *tokens* that are processed by a syntax analyzer, which is generated by *yacc* [Joh],

according to the grammar rules, producing a *parse tree* and a *symbol table*. These two steps are similar to scanning and parsing for conventional compilers, described in the literature [ASU85].

The main problem during this phase, as described in Section 4.1, is addressed by the symbol table manager with the differentiations between variables and functions. By the end of this phase, the symbol table manager will have created a list containing all the M-files invoked by the main program. This list is then used by the M-File inliner.

3.2.3 Inlining of M-files

Functions require a special treatment by the compiler. This problem is important because the knowledge of the statements inside a function might be necessary when defining the properties of its output parameters. Hence, functions cannot be treated as simple black boxes.

For the case of built-ins, we compile each function independently, using a database that contains all the necessary information about the function for the inference process. Hence, for a particular built-in, its 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 function compilation problem by *inlining* all the M-files that are used in the program, including multiple instances of the same M-file and multiple levels of M-file calls (i.e., M-files called inside another M-file). We then compile the entire program at once.

The M-file inliner is divided into two steps. The first step consists of a recursive procedure that traverses the list of M-Files generated by the symbol table manager and generates an independent parse tree and symbol table for each M-file in the list. These M-file's parse trees and symbol tables are generated using the same tools and functions described in Section 3.2.2. Each new M-file invoked by the M-file that is being parsed is added to the list of M-files. This step finishes when a parse tree and its corresponding symbol table have been created

for all M-files in the list. Notice that if the same M-file is called multiple times in the whole program, only one parse tree and corresponding symbol table are generated. As described below, however, this same parse tree will be inlined in the main program's parse tree multiple times.

The second step is the actual inlining. In this step, the main program's parse tree is read in lexicographic order and, for each call to an M-file, a copy of its corresponding parse tree is inserted in the main program's parse tree, replacing the M-file call. At this point, nodes are inserted in the parse tree to copy the actual parameters of the M-file call into the formal parameters of the M-file definition, and to copy the output parameters of the M-file into their corresponding variables in the program.

In contrast to regular Fortran or C functions, MATLAB functions can return more than one output variable and, although the number of output parameters in the function definition is fixed, the actual number of parameters returned by the function is dependent on the function call. Thus, the copy of the output parameters is performed only for the actual variables used by the M-file caller.

Finally, the M-file's symbol table is integrated into the program's symbol table, with all the M-file's local variables renamed to receive unique names. The reason for these unique variable names, as well as the multiple instances of the same M-file and copies of the M-file's input and output parameters, is that M-files are polymorphic, and their variables have local scope. Therefore, the same variable in an M-file may have different properties in different activations of the M-file, depending on the properties of the actual input parameters.

3.2.4 The Intermediate Representation

After the inline of the M-files, the parse tree is transformed into an *abstract syntax tree* (AST) [McK76], which has an intermediate representation more suitable for the generation of the output code.

Our inference algorithms are applied to a *Static Single Assignment* (SSA) [CFR+91]

```
S1:  load %(V)
S2:  n = length(V);
S3:  T = 0;
S4:  for k=1:n

S5:    T = T + V(k);
S6:  end

S7:  AVG = T / n;
         (a)
```

```
S1:  load %(V1)
S2:  n1 = length(V1);
S3:  T1 = 0;
S4:  for k1=1:n1
P1:    T4 = φ(T2,T1);
S5:    T2 = T4 + V1(k1);
S6:  end
P2:  T3 = φ(T2,T1)
S7:  AVG1 = T3 / n1
         (b)
```

Figure 3.3: MATLAB code to compute the average of a vector (a) and its respective SSA representation (b).

representation of the MATLAB program in the form of an AST. A program in SSA form has two main properties: first, each assignment creates a new instance of the variable; and each use of the variable can have only one possible assignment. Hence, in the SSA representation, each variable is assigned a value by at most one statement. When several definitions feed a single use of a variable, one or more ϕ function operators are inserted at the points of confluence in the control flow graph to merge the different definitions into a single variable. These ϕ functions are created and inserted into the AST during this phase. SSA is a convenient representation for our analysis algorithms because it is evident which definitions affect (or cover) a particular use of a variable.

Consider for example the MATLAB program to compute the average of a vector[1], as presented in Figure 3.3(a). In this example, there are two statements (S5 and S7) where variable `T` may have multiple definitions. For these cases, new instances of the variable are created and assigned according to the value of a ϕ function, as shown in Figure 3.3(b).

Each variable instance in the symbol table and each node of the AST representing a variable, a constant, or an operator contain the following attribute fields to store inference

[1]This is not the best way of writing a MATLAB program to compute the average of a vector. However, this form is more adequate for the presentation of the SSA representation.

information:

- number of rows;
- number of columns;
- rank;
- intrinsic type;
- structure;
- maximum estimated value;
- minimum estimated value;

Our inference mechanism considers only SCALARS, VECTORS, and two-dimensional MATRICES; hence, only two attributes are necessary for the representation of shape, namely number of rows and number of columns.

The attribute structure is used for the structural inference, as described in Section 5.5. Finally, the fields minimum and maximum estimated value are used to store and propagate the value information used by our value-propagation technique, described in Section 5.2.

All attribute fields, with the exception of the estimated values, are initialized as UNKNOWN. The maximum and minimum estimated values are initialized as $+\infty$ and $-\infty$ respectively. These attribute fields are filled during the inference phases and propagated through the AST and the symbol table whenever a new attribute is synthesized.

These attribute fields are used by the inference mechanism to synthesize information. In some cases, a node representing a variable in the AST and its corresponding instance in the symbol table may have the same values for their attribute fields. This, however, is not always the case. The node in the AST contains the information for the operand represented by the variable, while the attribute fields in the symbol table contain the information for the instance of the variable. Consider for example Statement S5 in Figure 3.3(b). The node

representing variable V_1 will contain the information that the rank of the operand is SCALAR and that its shape is (1,1), while its corresponding instance in the symbol table will contain the information that V_1 is a VECTOR with shape (1,?) or (?,1). This latter information is used in conjunction with the information from the AST node representing the array index (k) to synthesize the information for the AST node representing the operand $V_1(k_1)$.

3.2.5 Static Analysis

The variable properties are estimated during the static analysis phase using a forward/backward traversal scheme similar to the one described in Section 2.1. We will briefly describe next the overall strategy for the inference of each variable property, and the sources of information for the static inference mechanism.

Intrinsic Type Inference

Although MATLAB operates on REAL and COMPLEX values only, our static inference mechanism also considers INTEGER and LOGICAL values. That is, it considers all Fortran 90 intrinsic data types, except for CHARACTER. These types are inferred according to the following type hierarchy:

$$\text{LOGICAL} \prec \text{INTEGER} \prec \text{REAL} \prec \text{COMPLEX}.$$

In our compiler, intrinsic type inference could be avoided if all variables were considered as COMPLEX. This, however, would affect tremendously the performance of the code due to the number of additional arithmetic operations required by COMPLEX variables. Consider for example a matrix multiplication `C = A * B`, where "A" and "B" are square matrices of size 100×100. This matrix multiplication requires approximately 2 Mflops for type REAL, and 8 Mflops for type COMPLEX. Hence, the determination of the correct variable intrinsic type during compile time is very important for the efficiency of the code.

Shape and Rank Inference

Shape and rank are inferred for the static allocation of storage for the variables through declarations. Although MATLAB considers all variables two-dimensional arrays, the identification of SCALARS and VECTORS through rank inference is important to avoid the unnecessary use of indices and the over-dimensioning of variables that may cause poor memory and cache utilization. Moreover, it is also necessary to provide the ability to recognize SCALARS and VECTORS through rank inference because MATLAB has different semantics for certain operators, depending on the rank of the operands since each one requires a different translation to Fortran 90. Consider for example the MATLAB expression for the multiplication of two variables `A * B`. This can be an inner-product, an outer-product, a matrix multiplication, a matrix-vector multiplication, or a scalar multiplication (by another scalar, vector, or matrix), depending on the ranks of `A` and `B`. Due to performance and conformability issues, each of these possibilities requires a different method for the computation and, therefore, a different library function or operator is used for each case.

Shape inference is also important for the efficiency of the 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 several cases it is impossible to determine array sizes since they may be input data dependent (e.g., they may be read as an input for the program) and, thus, the only alternative is to allocate dynamically. In this case, an extra effort is necessary to predict the maximum size that an array will take in order to avoid the extra overhead of multiple execution time tests and reallocations of a variable with every change in shape. To this end, we developed a symbolic-propagation technique, discussed in Section 6.2.1.

Structural Inference

One of the great advantages of MATLAB is the encapsulation of its built-in functions and operators. For example, the linear equation

$$A \times x = b \tag{3.1}$$

can be solved in MATLAB using the simple statement:

```
x = A \ b;
```

When executing this statement, MATLAB performs the following tests [Mat92b] to choose the best method to solve the linear system:

- If A is a TRIANGULAR MATRIX, or a PERMUTATION of a TRIANGULAR MATRIX, then x is computed by a permuted back-substitution algorithm.
- If A is SYMMETRIC, or HERMITIAN, and has positive diagonal elements, then a Cholesky factorization is attempted.
- If A is SPARSE, then a symmetric minimum pre-ordering is applied.
- If A is SQUARE, and all the above tests fail, then a general triangular factorization is computed by Gaussian elimination with partial pivoting. If A is sparse, a non-symmetric minimum degree pre-ordering is applied.
- If A is a not square full matrix, then Householder reflections are used to compute an orthogonal-triangular factorization.
- If A is a not square sparse matrix, the least squares method is applied using sparse Gaussian elimination with numerical pivoting.

One of the challenges of the compiler is to exploit the high-level semantics of the language in order to detect some of these matrix structures statically, to avoid run-time tests, and to improve the performance of the target code.

A structural inference mechanism was designed to improve code performance by enhancing library selection by the code generator. Suppose for example, that in a matrix multiplication, both variables are known to be matrices, but one of them can be inferred to be a triangular matrix. In this case, instead of using a library function to perform a general matrix multiplication operation, an optimized function that takes into consideration the triangular structure of the variable could be used. This structural inference mechanism has not been implemented in the current version of the compiler.

Sources of Information for the Static Inference

The static inference mechanism extracts information from four main sources: input files; program constants; operators; and 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 rank[2] of the variables being loaded[3]. Variable shapes are not extracted from the input files because they are much more likely than intrinsic type and rank to differ between runs. However, with the rank information from the loaded variables, the compiler can propagate partial (or even complete) shape information represented in terms of values obtained when vectors (and scalars) are loaded.

The second source of information is program constants, from which intrinsic type, rank, and shape can be inferred and propagated. MATLAB operators are the third source of information. From them, we can extract type information in the case of logical operators, and rank, shape, and structural information by taking into consideration the conformability requirements imposed by the MATLAB operators.

Finally, the fourth source of information is the *high-level summary information* from the MATLAB intrinsic functions. Built-in functions can provide information for both forward and backward inference. An example of this is the function `rcond`, for the conditional

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

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

reciprocal estimator, which requires the input parameter to be a square matrix (useful for backward inference), and generates as output a scalar of type REAL (useful for forward inference). Most of the information for the structural inference is obtained from built-in functions.

3.2.6 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, the system generates code to determine the necessary attribute at run-time. We follow an approach similar to that used in many systems: associating tags with each variable, that after the static analysis, have UNKNOWN intrinsic type or shape. Based on these tags, which are stored in *shadow variables*, conditional statements are used to select the appropriate operations and to allocate the necessary space.

The dynamic phase is divided in the two steps. First, the AST is traversed to determine all the instances where dynamic type inference is necessary and the conditional statements for dynamic type inference are inserted, as described above. Second is the dynamic shape and rank inference, where dynamic code is generated to compute during run-time the necessary space for the dynamic allocation, and to select the appropriate method to perform certain operations. The overall strategy for each step of the dynamic inference is described next.

Dynamic Type Inference

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. The intrinsic type attribute in the corresponding nodes of the AST is updated accordingly.

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. For example, consider the following expression:

$$X = A \times B + C \times D \tag{3.2}$$

If A, B, C, and D are square matrices of size 100×100, each matrix multiplication will require approximately 2 Mflops for type REAL, and 8 Mflops for type COMPLEX. Thus, we observe that each multiplication can be computed up to four times faster if COMPLEX arithmetic is avoided with the inclusion of conditional statements.

It is important to observe that due to our static type inference mechanism, the generation of shadow tests for intrinsic type does not occur very often in practice. (The reason for this is explained in Section 5.1.) In our test cases using actual MATLAB programs, there were no situations where 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.

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`. Hence, for an expression with n operands, there will be at most $n-1$ triplets with at most 4 cases each ([REAL,REAL]; [COMPLEX,REAL]; [REAL,COMPLEX]; and [COMPLEX,COMPLEX]). That is, we consider at most $4(n-1)$ cases as opposed to the 2^n cases that would arise if the expression were not decomposed. The transformation into triplets is performed during the generation of the AST (Section 3.2.4).

```
if (A__T .eq.  T__COMPLEX) then
  if (B__T .eq.  T__COMPLEX) then
    T__1__C = A__C * B__C
  else
    T__1__C = A__C * B__R
  end if
  T__1__T = T__COMPLEX
else
  if (B__T .eq.  T__COMPLEX) then
    T__1__C = A__R * B__C
    T__1__T = T__COMPLEX
  else
    T__1__R = A__R * B__R
    T__1__T = T__REAL
  end if
end if
T__2 = C * D
if (T__1__T .eq.  T__COMPLEX) then
  X__C = T__1__R + T__2
  X__T = T__COMPLEX
else
  X__R = T__1__R + T__2
  X__T = T__REAL
end if
```

Figure 3.4: Dynamic code for `X=A×B+C×D`, when `A` and `B` have UNKNOWN type.

Determining on which case should be used during execution time is dependent on the value of the corresponding shadow variable for the intrinsic type. For example, consider again Expression (3.2) and suppose, that the compiler was not able to infer the variable intrinsic types of A and B, and that the types of C and D are known to be REAL.[4] The pseudo-code generated for this case is presented in Figure 3.4, where the suffixes `__R` and `__C` represent the REAL and COMPLEX definitions for a variable, and shadow variables for intrinsic type are represented by the `__T` suffix. Notice that the multiplication $A \times B$ is generated in four different ways, depending on the dynamic type of the operands.

[4]If any of these variables were inferred as type COMPLEX, then the resulting variable would also be of type COMPLEX; however, the conditional statements would still be necessary when A and B are REAL.

```
S1: Ap = A * p;
S2: alpha = (r' * p) / (p' * Ap);
S3: x = x + alpha * p;
S4: r = r + alpha * Ap;
```

Figure 3.5: MATLAB code segment to compute conjugate gradient.

A simpler approach would be to consider all variables that cannot have their type inferred statically by the compiler as having intrinsic type COMPLEX One problem with this approach is that for each of these undefined variables, the type will have to be propagated as a COMPLEX variable, and this propagation will tend to perturb the inference process. For example, consider the MATLAB code segment from a function that computes conjugate gradient, as presented in Figure 3.5.

Suppose that the type of `A` is UNKNOWN and that the compiler assumes it to be of type COMPLEX. Assume also that all other variables are known to be of type REAL. We notice that the variable `A` appears only once in this code segment; however, the compiler will have to consider `Ap`, `alpha`, `x`, and `r` as COMPLEX variables because of its assumption about the type of `A`.

The generation of dynamic code is avoided whenever there is an assignment that redefines a variable, changing its intrinsic type. To this end, the variable is renamed and a second declaration for the variable is generated. Thus, for example, for the pseudo-code presented in Figure 3.6, the compiler will declare statically three instances of the variable, one for each intrinsic type assumed by the variable, and generate the code as shown in figure 3.7.

Dynamic Shape and Rank Inference

The second step of the dynamic phase is the dynamic shape and rank inference. During this step, dynamic code is generated to compute during run-time the necessary space for the dynamic allocation. To this end, two shadow variables are used to keep track of variable dimensions during execution-time. So, for example, if the shape of `B` in an assignment of

```
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);
```

Figure 3.6: Pseudo-code example where the intrinsic type of the variable changes between assignments.

```
    INTEGER temp_1
    COMPLEX*16 temp_2
    DOUBLE PRECISION temp_3
    ...
S1: temp_1 = 1
S2: X = function(temp_1)
    ...
S3: temp_1 = 10
S4: W = function(temp_1)
    ...
S5: temp_2 = sqrt(-1)
S6: Z = function(temp_2)
    ...
S7: temp_3 = 0.5
S8  Y = function(temp_3)
```

Figure 3.7: Fortran 90 code for multiple assignments with different intrinsic types to the same variable.

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

Figure 3.8: Example of shadow variables for shape.

the form `A=B+0.5` were UNKNOWN at compile time, the compiler would generate the code presented in Figure 3.8, which uses shadow variables `A_D1`, `A_D2`, `B_D1`, and `B_D2`, to store the run-time information about the dimensions 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. So, for example, if this is the first definition of variable `A`, statements `S1` and `S2` are not generated[5].

Dynamic rank inference is also generated for the correct execution of the code. Consider again a multiplication of two variables being assigned to a third variable: `c = a` $*$ `b`. Suppose that both variables `a` and `b` are of type REAL, but have UNKNOWN shape at the end of the static analysis phase.

The simple solution of generating a single matrix multiplication statement for this expression (e.g., `c = MATMUL(a,b)`) is not possible due to performance and conformability considerations. Thus, dynamic rank inference is performed in order to select the appropriate functions to perform the operations according to the rank of `a` and `b`. The resulting Fortran 90 code would be the code sequence[6] shown in Figure 3.9.

The shadow variables of `a` and `b` are used to infer the rank of the variables during execution time. The shadow variable for `c` is updated according to the operation that is

[5]If this definition occurs inside a loop, it is not considered the first definition because there will be an artificial assignment to NULL outside the loop, as described in Section 4.2.2.

[6]The compiler assumes the correctness of the original program. It uses the BLAS routines `DDOT`, `DGEMV`, and `DGEMM` for dot-product, matrix-vector multiplication, and matrix multiplication respectively.

```
if (a_D1 .eq. 1) then
  if (a_D2 .eq. 1) then
    if (c_D1 .ne. b_D1 .or. c_D2 .ne. b_D2) then
      if ( ALLOCATED(c) ) DEALLOCATE(c)
      c_D1 = b_D1
      c_D2 = b_D2
      ALLOCATE(c(c_D1,c_D2))
    end if
    c = a(1,1) * b
  else
    if (b_D1 .eq. 1) then
      if (c_D1 .ne. a_D1 .or. c_D2 .ne. a_D2) then
        if ( ALLOCATED(c) ) DEALLOCATE(c)
        c_D1 = a_D1
        c_D2 = a_D2
        ALLOCATE(c(c_D1,c_D2))
      end if
      c = a * b(1,1)
    else
      if (b_D2 .eq. 1) then
        if (c_D1 .ne. 1 .or. c_D2 .ne. 1) then
          if ( ALLOCATED(c) ) DEALLOCATE(c)
          c_D1 = 1
          c_D2 = 1
          ALLOCATE(c(c_D1,c_D2))
        end if
        c = DDOT(a_D2, a, 1, b, 1)
      else
        if (c_D1 .ne. 1 .or. c_D2 .ne. b_D2) then
          if ( ALLOCATED(c) ) DEALLOCATE(c)
          c_D1 = 1
          c_D2 = b_D2
          ALLOCATE(c(c_D1,c_D2))
        end if
        CALL DGEMV('C', b_D1, b_D2, 1.0D0, b, b_D1, a,1 , 0.0D0, c, 1)
      end if
    end if
  end if
else
  if (b_D2 .eq. 1) then
    if (b_D1 .eq. 1) then
      if (c_D1 .ne. a_D1 .or. c_D2 .ne. a_D2) then
        if ( ALLOCATED(c) ) DEALLOCATE(c)
        c_D1 = a_D1
        c_D2 = a_D2
        ALLOCATE(c(c_D1,c_D2))
      end if
      c = a * b(1,1)
    else
      if (c_D1 .ne. a_D1 .or. c_D2 .ne. 1) then
        if ( ALLOCATED(c) ) DEALLOCATE(c)
        c_D1 = a_D1
        c_D2 = 1
        ALLOCATE(c(c_D1,c_D2))
      end if
      CALL DGEMV('N', a_D1, a_D2, 1.0D0, a, a_D1, b,1 , 0.0D0, c, 1)
    end if
  else
    if (c_D1 .ne. a_D1 .or. c_D2 .ne. b_D2) then
      if ( ALLOCATED(c) ) DEALLOCATE(c)
      c_D1 = a_D1
      c_D2 = b_D2
      ALLOCATE(c(c_D1,c_D2))
    end if
    CALL DGEMM('N', 'N', a_D1, b_D2, a_D2, 1.0D0, a, a_D1, b, b _D1, 0.0D0, c, c_D1)
  end if
end if
```

Figure 3.9: Fortran 90 code for `c=a*b` when rank and shape of `a` and `b` are UNKNOWN.

being performed (e.g., multiplication, left solve, right solve) and the shape of `a` and `b`. They are then used to guide the allocation (or reallocation) of the array `c`, whenever necessary.

Notice that special consideration is taken with SCALARS when the variable is dynamically allocated. The problem is that the Fortran 90 operation `a * b` assumes the operands to be conformable if they are dynamically allocated. Hence, if one of the operands is SCALAR and if it is dynamically allocated as a 1×1 array, the operation as written above would result in an error. Therefore, dynamic rank inference is necessary. As presented in Section 6.2.1, this dynamic code can be optimized with the use of symbolic-propagation.

3.2.7 Code Generator

The final phase of the MATLAB compiler generates Fortran 90. This pass is also divided into two steps: the generation of declarations, and the generation of executable code.

The declarations are generated by traversing the symbol table. Variables that have UNKNOWN intrinsic type at the end of the static phase are declared twice: once for the REAL instance of the variable, and once for the COMPLEX instance. Variables that have UNKNOWN shape are declared as two-dimensional allocatable arrays. Shadow variables are generated only for variables that have UNKNOWN shape or intrinsic type at the end of the static phase.

The generation of executable code is performed by traversing once more the AST in lexicographic order. This pass is straightforward because our output language is a high-level language and, therefore, we do not deal with most of the issues of code generation [ASU85]. Moreover, at this point, all necessary attributes for the code generation have been already filled by the static and dynamic inference phases.

To maintain compatibility with the MATLAB interpreter, whenever possible[7] the compiler generates function calls from the same libraries used by MATLAB (i.e., LINPACK [DMBS79], LAPACK [ABB+92], EISPACK [SBD+76]).

[7] The information on which library function is being used by the interpreter is not always available. Also, sometimes this information is provided with a note stating that the library function was modified.

Chapter 4

INTERNAL REPRESENTATION

4.1 Distinguishing Variables from Functions in MATLAB

In MATLAB, an identifier occurrence may represent a function or a variable, depending on the context in which the identifier is first referenced and the files present in the execution environment. For example, consider the pseudo-code presented in Figure 4.1. The identifier "`i`" in Statement S2 is a built-in function that returns the imaginary unit, while in Statements S3 and S4 it represents the variable "`i`" used as the loop index.

In general, a MATLAB identifier may represent a function instead of a variable if, in the lexicographic order of the program, it appears on the right hand side (RHS) before it appears on the left hand side (LHS). If the identifier is not a valid built-in function, the files in the execution environment determine whether the identifier represents an M-file. An example of this is presented in Figure 4.2, which contains two versions of pseudo-code to compute the cumulative sum of a vector.

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

Figure 4.1: MATLAB pseudo-code segment in which the same expression has different semantics.

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

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

Figure 4.2: Two MATLAB code segments to compute the cumulative sum of a vector.

The execution of both code segments in Figure 4.2 will generate the same results if there is no M-file "`s.m`" defined in the user's path or in MATLAB's path. However, code segment (a) will generate wrong results (or may generate an error) if such a file exists. The reason for this inconsistent behavior is that MATLAB parses the program in lexicographic order. If it 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 "`s`" in code segment (a) appears on the RHS, MATLAB will execute the program correctly only if there is no "`s.m`" M-file. However, if such an M-file is found, MATLAB will execute this function every time it executes Statement S4. The code segment in Figure 4.2(b) will always generate the correct result because the first reference to the variable "`s`" appears on the LHS.

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

Algorithm: *Differentiation Between Variables and Functions*

Input: A MATLAB program and its correspondent symbol table.

Output: The *kind* of each identifier in the program; that is, whether the identifier represents a variable, a function, or both throughout the program.

This algorithm is based on the state diagram presented in Figure 4.3. Each identifier can be in one of the following states: Built-in, M-file, Empty, Function, Variable, or Multiple. The Start state is responsible for the definition of the initial state for each identifier. This definition is based on one of the following three conditions:

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

After defining the initial state of all identifiers, the MATLAB program is traversed in lexicographic order. Every identifier occurrence (*read* or *write*) causes a transition in the state diagram. The state at the end of the program is the kind of the identifier. An identifier will be in the state Multiple if at some point in the program it represented a function (Built-in or M-file) and became a variable later in the program. An example of an identifier of the kind Multiple is "`i`" in Figure 4.1.

End *algorithm for differentiation between variables and functions.*

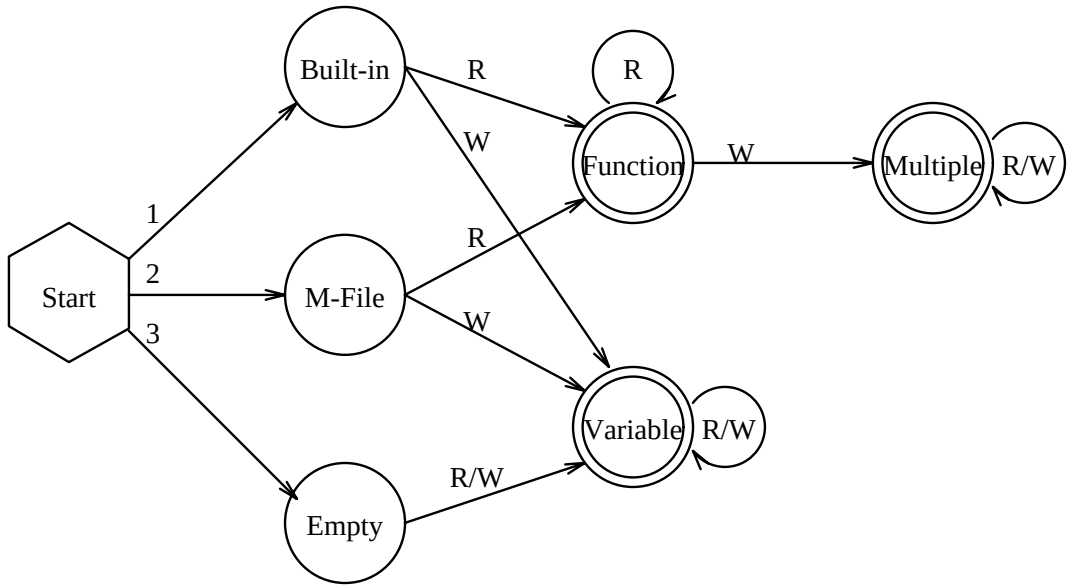

Figure 4.3: State diagram for differentiation between functions and variables.

4.2 Static Single Assignment

4.2.1 Extensions to the Symbol Table for the SSA Representation

To support the SSA representation, our symbol table is divided in two main parts: the *entry list* and for each entry, a list containing all *variable instances*. As an example, consider again the SSA representation of the program to compute the average of a vector, presented in Figure 3.3(b). The data structure for the symbol table corresponding to this program is outlined in Figure 4.4.

Each node in the entry list contains the information that is shared by all instances of the corresponding variable, such as loaded name and program name[1], as well as global information used by the symbol table manager, such as the number of instances and a pointer to the start of the corresponding variable's instance list. Information about built-ins and M-files are also stored in the symbol table. Therefore a field *kind* is used to indicate if the entry is a variable, built-in, M-file, or multiple (for identifier in the state Multiple described

[1]The original MATLAB name must be saved for the `load` function. The loaded name may be different than the program name due to renaming performed by the inliner.

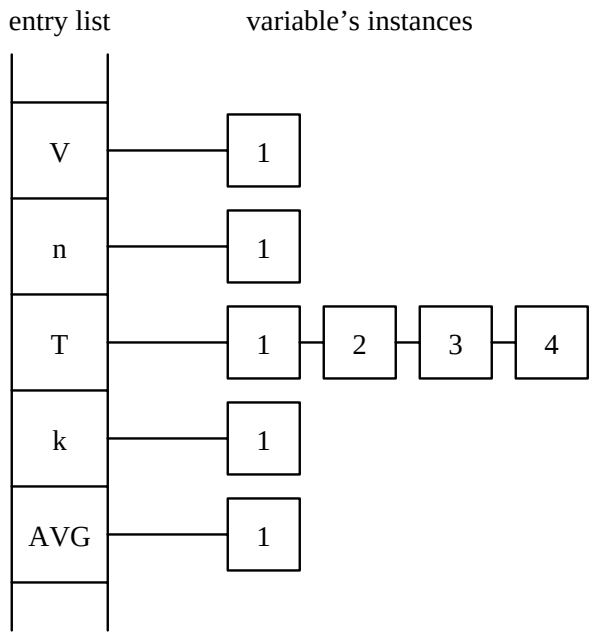

Figure 4.4: Symbol table for the SSA representation.

above).

Each node in the variable instances list contains information specific to a particular instance of the variable, such as the set of attribute fields (described in Section 3.2.4), its index number, and a pointer to the variable's definition node in the AST.

4.2.2 SSA Representation for Scalar and Full Array Assignments

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(R)=RHS`, where `R` is a *range* that can be a scalar, a vector, or a two-dimensional array represented with constants, variables, or both). Full array assignments can be treated as scalar assignments since each full array assignment completely redefines its previous definition.

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 two control arcs, which means ϕ functions always have two parameters. Each parameter can have a *backward definition* if it corresponds to

a variable previously defined in lexicographic order, or a *forward definition* if it points to a variable definition that post-dominates the ϕ function.

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 redefine as λ functions, replaces the ϕ functions required at the end of conditional statements and loops. Both parameters of λ functions have backward definitions. The other group of ϕ functions, which we redefine as φ functions, replaces the ϕ functions required at the beginning of loops. φ functions have one forward definition and one backward definition.

In some cases, the use of a variable may not have a previous definition in the lexicographic order of the program, as for example variable `y` in Statement S3 in the code segment presented in Figure 4.5(a). In these cases, when generating the λ and φ functions, the corresponding backward definitions are assumed to be NULL, as exemplified by Statement P1 in Figure 4.5(b) that presents the SSA representation for the same program. Thus, as shown in Statement P2, the φ function will have two parameters; however, one points to an "*actual*" assignment, while the other points to an "*artificial*" assignment to NULL. These artificial assignments are not necessary for the static inference mechanism, as the φ function could have just the forward parameter. However, its use facilitates the implementation by forcing all φ functions to have two parameters. It is also helpful during the dynamic phase, as discussed in Chapter 6.

4.2.3 Extension of the SSA Representation to Support Indexed Array Assignments

While 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 approach involving a new function. Consider, for example, the MATLAB code segment presented in Figure 4.6, and assume that `A` is assigned in statements S2 and S4 only. In this code segment, S1 creates a matrix `Z` filled with zeros; S2 creates a matrix

```
S0: load %(n)

S1: for k=1:n

S2:   if (k > 1)
S3:     z = y / k;
S4:   else
S5:     z = k1;
S6:   end

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

(a)

```
S0: load %(n1)
P1: y0 = ∅
S1: for k1=1:n1
P2:   y2 = φ(y1,y0)
S2:   if (k1 > 1)
S3:     z1 = y1 / k1;
S4:   else
S5:     z2 = k1;
S6:   end
P3:   z3 = λ(z1,z2)
      ...
S7:     y1 = z3;
S8: end
```

(b)

Figure 4.5: Example of a variable use and definition not in lexicographic order.

A filled with REAL numbers; and S4 updates a range of A with values from Z. As previously mentioned, there is no problem in treating Statements S1 and S2 as scalar assignments. They generate an INTEGER and REAL array respectively. Moreover, if statement S4 did not involve subscripts and were of the form A = Z, the SSA representation would be obtained by simply renaming A in S2 and S4 and the corresponding uses. However, since S4 updates only a part of A, a simple renaming of A would not be correct. If we use the standard SSA approach and consider that the integer assignment to A in S4 will generate a new instance of the variable with type INTEGER, the type inference would generate wrong results. Hence, in situations like this, we use the standard approach and transform indexed assignments of the form A(R) = RHS; where R is an arbitrary range, into

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

The α function we use is similar to the "**Update**" function described in [CFR+91], but extended for assignments to a matrix range. In the case of MATLAB, an α function may return an array with dimensions larger than that of the parameter array. In this case, if the array section R completely covers the old range of A, the values and shape of A are those of

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

Figure 4.6: MATLAB code with indexed array assignment.

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

Figure 4.7: SSA representation for indexed array assignments.

the RHS. Otherwise, array A will have to be expanded in one or both dimensions[2], and the unassigned elements are given the value zero. Using this extension, the SSA representation for the code presented in Figure 4.6 will be as shown in Figure 4.7, where the assignments to A have been renamed and merged in the same manner as for scalar assignments.

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

Chapter 5

THE STATIC INFERENCE MECHANISM

In this chapter we discuss the main topics on the static inference mechanism, starting with description of the algorithms for the intrinsic type inference, followed by the value-propagation technique used to improve the intrinsic type inference, the shape and rank inference mechanism, a discussion about compilation of functions, and finally, a description of the structural inference. For clarity of presentation, the intrinsic type inference, and shape and rank inference are discussed separately, despite the fact that they are applied by a single compiler pass.

5.1 Intrinsic Type Inference

The static mechanism for intrinsic type inference propagates intrinsic types through expressions using a *type algebra* similar to that described in [Sch75] for SETL. For the case of logical operators, the output 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. These tables contain for each operation 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 promotes 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 presented in Section 3.2.5. In some cases, however, (e.g., "\" left and "/" right division) the output will be 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 result in an output of intrinsic type REAL). For the cases where the outcome of an expression can be of different intrinsic types, depending on the values of the operands (such as the power operator, square root, and inverse trigonometric functions), we use our value-propagation technique, described in Section 5.2, to infer statically the range of values of the variables. When this value-based analysis fails, we promote the output type of the expression to COMPLEX.

The general algorithm for intrinsic type inference is described below.

Algorithm: *Intrinsic Type Inference*

Input: An SSA representation of a MATLAB program (or segment) in the form of an AST.

Output: The intrinsic type for all variables in the program (or segment).

The algorithm traverses the AST in lexicographic order. Its actions are dependent on the kind of each statement visited, as described next. In all cases, after the action is taken, the algorithm proceeds with the analysis of the next statement.

- Control Flow: Do nothing.
- Input (i.e., `load`): Assign the intrinsic type of all loaded variables according to the information from the loaded file.
- Assignment to an expression or constant: Assign the intrinsic type of the output according to the type algebra.
- Assignment to a built-in function: Assign the intrinsic type(s) of the output(s) according to the database for built-in functions.

Type of first parameter	Type of second parameter					
	NULL	LOGICAL	INTEGER	REAL	COMPLEX	UNKNOWN
NULL	NULL	LOGICAL	INTEGER	REAL	COMPLEX	UNKNOWN
LOGICAL	LOGICAL	LOGICAL	INTEGER	REAL	UNKNOWN	UNKNOWN
INTEGER	INTEGER	INTEGER	INTEGER	REAL	UNKNOWN	UNKNOWN
REAL	REAL	REAL	REAL	REAL	UNKNOWN	UNKNOWN
COMPLEX	COMPLEX	UNKNOWN	UNKNOWN	UNKNOWN	COMPLEX	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

Table 5.1: Resulting type for the comparison of λ or φ functions parameters.

- Assignment to a λ function: Compare the intrinsic types of the parameters and assign the intrinsic type of the output according to Table 5.1.
- Assignment to a φ function: Execute the iterative algorithm for intrinsic type inference of φ functions, as described in Section 5.1.1.
- Assignment to an α function: Execute the iterative algorithm for intrinsic type inference of α functions, as described in Section 5.1.2.

End *algorithm for intrinsic type inference.*

As can be observed in the algorithm for intrinsic type inference, the type inference for λ functions is straightforward because both parameters have backward definitions. However, special consideration is necessary in the presence of φ and α functions. The following sections describe the algorithms for propagation of intrinsic type information through φ and α functions.

5.1.1 Propagation of Intrinsic Type Information Through φ Functions

The static inference of a φ function requires an iterative process because the forward definition has yet to be analyzed when it is encountered for the first time. To this end, the following algorithm is applied for the inference of the intrinsic type of φ functions.

Algorithm: *Intrinsic Type Inference of φ Functions*

Input: An assignment to a φ function of the form:

$$A_v = \varphi(A_f, A_b).$$

Output: The intrinsic type of the variable A_v, assigned according to the output of a φ function.

This algorithm assumes A_f and A_b to be respectively the forward and backward definitions of the variable A, and A_f__Type and A_b__Type to be their respective intrinsic type attribute. The intrinsic type attribute of the φ function is assigned to A_v__Type that is used whenever the variable A_v is used. The algorithm also uses a temporary variable (A_t__Type).

1. A_f__Type $\Leftarrow$ NULL
2. A_t__Type $\Leftarrow$ A_b__Type
3. *DO*
4. A_v__Type $\Leftarrow$ A_t__Type
5. *EXECUTE* the intrinsic type inference algorithm from the statement following the φ function in the AST until the definition of A_f
6. *INFER* A_f__Type
7. *INFER* A_t__Type as the output of the φ function, according to Table 5.1
8. *UNTIL* A_t__Type == A_v__Type.

End *algorithm for intrinsic type inference of φ functions.*

This iterative process is guaranteed to finish because, as shown in Figure 5.1, our inference system for λ and φ functions uses a type lattice that is finite. Hence, each new iteration of the process will consider a new type at least one level higher in the lattice. Thus, in the worst case, the output of the φ function and all variables that directly or indirectly use

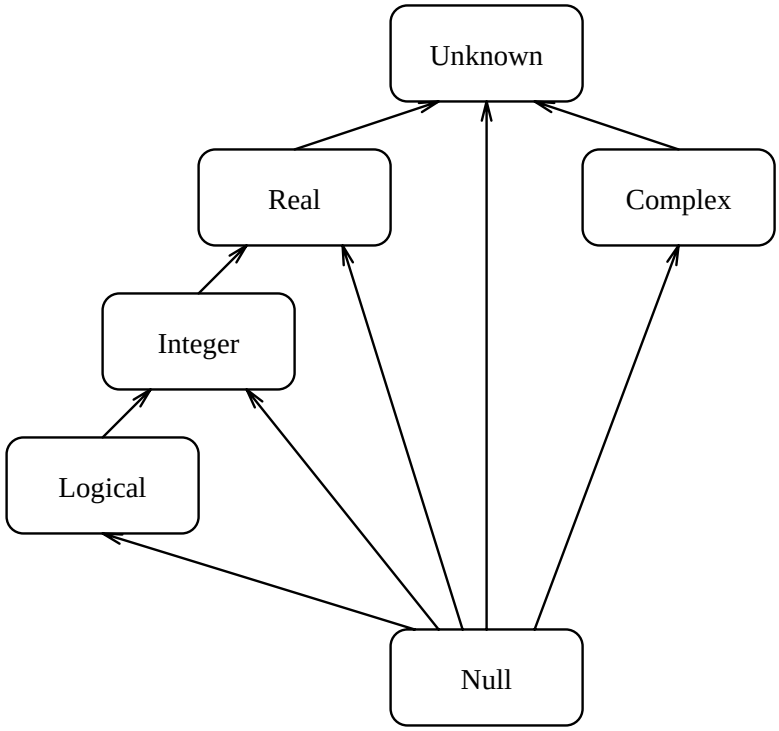

Figure 5.1: Intrinsic type lattice for φ functions.

this output will be considered as UNKNOWN, and the process will finish. Note that because NULL is an artificial type, the combination of NULL with any other intrinsic type results in the non-NULL type. Notice also that in Table 5.1, the confluence of REAL and COMPLEX is UNKNOWN rather than COMPLEX. In this way, if a variable can contain both REAL and COMPLEX values at execution time, COMPLEX operations will be executed only when the variable have a COMPLEX value.

5.1.2 Propagation of Intrinsic Type Information Through α Functions

For the intrinsic type inference of α functions, the following iterative algorithm is applied:

Algorithm: *Intrinsic Type Inference of α Functions*

Input: An assignment to an α function of the form:

$$A_i = \alpha(A_p, \mathrm{R}, \mathrm{RHS}).$$

Type of RHS	Type of previous definition					
	EMPTY	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

Table 5.2: Resulting type for α functions.

Output: The intrinsic type of the variable A_i, assigned according to the output of an α function.

This algorithm uses the last full definition of A (A_{full}), previous to the definition of A_i. If such a definition does not exist, the algorithm considers A_{full} to be an artificial definition assigned to NULL, before the first definition of A.

1. *INFER* A_i_Type according to Table 5.2
2. *IF* (A_i_Type $\neq$ A_p_Type) *THEN*
3. *REPEAT*
4. A_p_Type $\Leftarrow$ A_i_Type
5. *EXECUTE* the intrinsic type inference algorithm from the statement following the definition of A_p in the AST until the definition of A_i
6. *INFER* A_i_Type according to Table 5.2
7. *UNTIL* (A_i_Type == A_p_Type)
8. *END IF.*

End *algorithm for intrinsic type inference of α functions.*

As can be observed in Table 5.2, to avoid the overhead of copying the array during execution time, whenever the intrinsic type of the RHS differs from that of the previous definition, we promote the resulting α function type to one that subsumes both types. Thus,

```
A = zeros(5,1);
for ...
  ...
  A(1:5) = COMPLEX expression;
  ...
```

Figure 5.2: Variable initialization using a lower intrinsic type.

if an α function output is inferred to be of a more general type than its previous definition, then the intrinsic type of the previous definition is set with this more general type. Therefore, the backward inference process is necessary to update the previous full assignment to the variable, the subsequent indexed assignments and the variables that use it.

To be able to detect possible initializations to a lower intrinsic type than those used by the indexed assignment, as for example in the pseudo-code segment presented in Figure 5.2, this algorithm is performed for all α functions, even if it can be proved statically that the range R covers the previous definition of the variable (A_p).

This algorithm is also guaranteed to finish for the same reason as the algorithm for intrinsic type inference of φ functions. In this algorithm, in the two worst cases, the participating variables will have intrinsic type COMPLEX or UNKNOWN. However, due to the decision of promoting the intrinsic types, the variables will have intrinsic type UNKNOWN only if the RHS or the previous definition (A_p) have intrinsic type UNKNOWN.

5.2 Value Propagation

In some cases the outcome of a built-in function, such as `sqrt` or `log`, can be of different types, depending on the values of the operands. From now on, these class of functions will be referred to as *multi-typed built-in* to emphasize the fact that their output 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.

Our standard intrinsic type inference for built-in functions which, as described in Section 3.2.3, uses a database containing the possible intrinsic types for the output variables based on the intrinsic types of the input parameters, is not the best approach for multi-typed built-ins. The inefficiency results because the same activation of the built-in may result in different intrinsic types, depending on the value of the parameters. The simplest approach for these kinds of functions, as used by MathWorks in their compiler [Mat95], is to assume that the output of these built-ins will always be of type COMPLEX. However, this automatic promotion of intrinsic type may affect the performance of the code.

To improve our type inference mechanism, we developed the following value-based analysis algorithm that takes into consideration the range of possible values for each variable in the program.

Algorithm: *Value-based Analysis*

Input: SSA representation of a MATLAB program in the form of an AST.

output: Estimated range of possible values (minimum and maximum) for all variables in the program.

The algorithm traverses the AST in lexicographic order, estimating the minimum and maximum possible values for each variable. For non-SCALAR variables, the estimated values correspond to the minimum and maximum possible values for all elements of the VECTOR or MATRIX. 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. The actions of the algorithm are dependent on the kind of each statement visited, as described next. In all cases, after the action is taken, the algorithm proceeds with the analysis of the next statement.

- Control Flow: Do nothing.
- Input (i.e., `load`): Attribute the range $-\infty$ to $+\infty$ for all loaded variables.

- Assignment to a constant: Attribute the constant as minimum and maximum values for the variable.
- Assignment to a built-in function that returns a fixed range (e.g., `sin`, `cos`, `norm`, `rand`): Attribute the output range of the function to the variable, if the parameter is LOGICAL, INTEGER, or REAL.
- Assignment to an expression:
  - If possible, evaluate the expression using the minimum and maximum estimated values of the operands, taking into consideration possible sign changes due to the expression. For example, the minimum and maximum estimated values for the expression x^2, considering that the range of x is $-\infty$ to 1, will be 0 and $+\infty$ respectively.
  - If not possible, attribute the range $-\infty$ to $+\infty$ to the output variable.
- Assignment to a λ function: Attribute the minimum and maximum estimated values of the two parameters.
- Assignment to an α function: If the range of estimated values of the RHS is larger than the range of estimated values of the previous definition of the variable, update its range, and execute an algorithm similar to the algorithm for intrinsic type inference of α functions.
- Assignment to a φ function: Execute an algorithm similar to the algorithm for intrinsic type inference of φ functions.

End *algorithm for value-based analysis.*

As an example, consider the following expression that is used for the Dirichlet solution to Laplace's equation:

$$\omega = \frac{4}{2 + \sqrt{4 - \left[cos\left(\frac{\pi}{n-1}\right) + cos\left(\frac{\pi}{m-1}\right)\right]^2}} \tag{5.1}$$

This expression returns an optimal choice for the parameter ω used to perform the successive over-relaxation (SOR). According to the SOR method, this parameter should be in the range $1 \leq \omega \leq 2$. 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 a 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.

Using our value-based analysis and considering that the types of n and m are known to be INTEGER (or REAL) values, the compiler can infer that the cosine function (*cos*) will return minimum and maximum values equal to -1 and 1 respectively. Thus, the addition of the two cosines will return a value between -2 and 2, and its square will result in a value between 0 and 4. Hence, the argument of the square root will be always ≥ 0, and ω can be inferred to be a REAL variable in the range $1 \leq \omega \leq 2$.

5.3 Shape and Rank Inference

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

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

Exclusive rank: For variables of which only one dimension is known. If the known dimension is 1, the variable is inferred to have a NOTMATRIX rank. Otherwise, if it is known to be greater than 1, the variable is inferred to have a NOTSCALAR rank.

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

Although compiled languages, such as C and Fortran, do not differentiate between row vectors and column vectors, we need to make this distinction because MATLAB does so.

A * B	B			
A	SCALAR	VECTOR(p,1)	VECTOR(1,q)	MATRIX(p,n)
SCALAR	SCALAR	VECTOR(p,1)	VECTOR(1,q)	MATRIX(p,n)
VECTOR(p,1)	VECTOR(p,1)	error	MATRIX(p,q)	error
VECTOR(1,q)	VECTOR(1,q)	SCALAR[1]	error	VECTOR(1,m)[1]
MATRIX(n,q)	MATRIX(n,q)	VECTOR(n,1)[1]	error	MATRIX(n,m)[1]
[1]Only if p = q; otherwise error.				

Table 5.3: Exact rank and shape inference for the multiplication operator.

MATLAB overloads operators with different semantics depending on the kind of vector that is being used. Variables that are not inferred to have an exact rank are marked to be dynamically allocated and, as mentioned before, are always defined as two-dimensional allocatable arrays.

The MATLAB operators can be divided in two main groups, depending on their conformability requirements. 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 shape (e.g., `+`, `-`, and logical operators) as a *shape conformable operator* the

Rank and shape information are obtained with the use of tables, such as Table 5.3 for the (exact) rank and shape inference for the multiplication operator, and Table 5.4 for the shape inference of a shape conformable operator. In these tables the shape information is indicated with the letters (m, n, p, and q) that represent the exact values for the number of rows or the number of columns; during the static phase of the analysis, shape information is obtained only from constants. COLUMNVECTORS and ROWVECTORS are represented as VECTOR(p,1) and VECTOR(1,q) respectively, and UNKNOWN values are represented with "?".

Although not implemented in the current version of the compiler, the conformability constraints imposed by the MATLAB language allow backward inference in several cases. Consider for example the addition of a variable `A` that is known to be a COLUMNVECTOR, with a variable `B` that is inferred to be NOTMATRIX with shape (1,?). As presented in Table 5.4 the output must be a COLUMNVECTOR; however, we can also infer that for

A+B	B								
	SCALAR	VECTOR		NOTMATRIX		NOTSCALAR		MATRIX	UNKNOWN
A	(1,1)	(p,1)	(1,q)	(1,?)	(?,1)	(p,?)	(?,q)	(p,q)	(?,?)
(1,1)	(1,1)	(p,1)	(1,q)	(1,?)	(?,1)	(p,?)	(?,q)	(p,q)	(?,?)
(p,1)	(1,1)	(p,1)	error	(p,1)	(p,1)	(p,1)	error	error	(p,1)
(1,q)	(1,q)	error	(1,q)	(1,q)	(1,q)	error	(1,q)	error	(1,q)
(1,?)	(1,?)	(p,1)	(1,q)	(1,?)	(?,?)	(p,?)	(?,q)	(p,q)	(?,?)
(?,1)	(?,1)	(p,1)	(1,q)	(?,?)	(?,1)	(p,?)	(?,q)	(p,q)	(?,?)
(p,?)	(p,?)	(p,1)	error	(p,?)	(p,?)	(p,?)	error	(p,q)	(p,?)
(?,q)	(?,q)	error	(1,q)	(?,q)	(?,q)	error	(?,q)	(p,q)	(?,q)
(p,q)	(p,q)	error	error	(p,q)	(p,q)	(p,q)	(p,q)	(p,q)	(p,q)
(?,?)	(?,?)	(p,1)	(1,q)	(?,?)	(?,?)	(p,?)	(?,q)	(p,q)	(?,?)

Table 5.4: Shape inference for a conformable operator.

A+B	B							
	VECTOR		NOTMATRIX		NOTSCALAR		MATRIX	UNKNOWN
A	(p,1)	(1,q)	(1,?)	(?,1)	(p,?)	(?,q)	(p,q)	(?,?)
(p,1)			B⇐(1,1)		B⇐(p,1)			B⇐(?,1)
(1,q)				B⇐(1,1)		B⇐(1,q)		B⇐(1,?)
(1,?)	A⇐(1,1)				A⇐(1,1)		A⇐(1,1)	
(?,1)		A⇐(1,1)				A⇐(1,1)	A⇐(1,1)	
(p,?)	A⇐(p,1)		B⇐(1,1)				A⇐(p,q)	
(?,q)		A⇐(1,q)		B⇐(1,1)			A⇐(p,q)	
(p,q)			B⇐(1,1)	B⇐(1,1)	B⇐(p,q)	B⇐(p,q)		
(?,?)	A⇐(?,1)	A⇐(1,?)						

Table 5.5: Backward inference for a conformable operator.

the program to perform correctly, the variable B must be SCALAR. Table 5.5 presents all situations where backward inference is possible for a shape conformable operator (+).

5.3.1 Propagation of Rank and Shape Information Through Both λ and φ Functions

The algorithms for propagation of rank and shape information through λ and φ functions are very similar to the algorithms for type inference described in Section 5.1. During the static phase of the analysis, shape information is obtained only from constants; hence, our lattice for shape inference, presented in Figure 5.3, is similar to the one described by Wegman and Zadeck in [WZ91] for constant propagation. For each dimension of a variable, there are

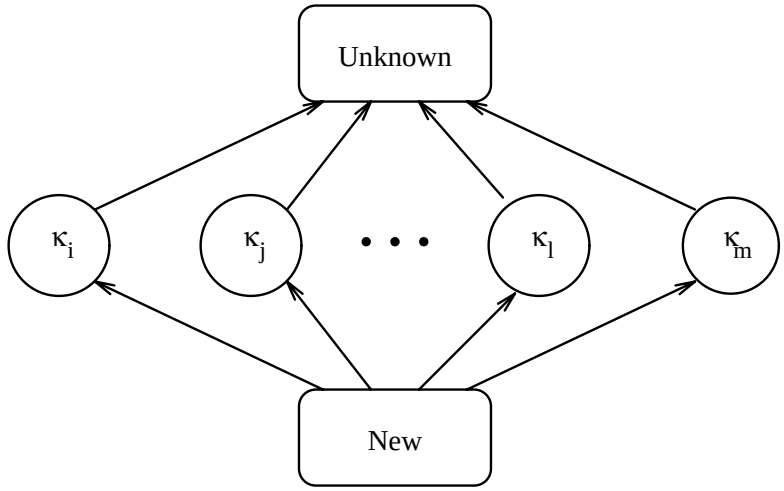

Figure 5.3: Lattice for static shape inference.

First parameter	Second parameter		
	NEW	κ_j	UNKNOWN
NEW	NEW	κ_j	UNKNOWN
κ_i	κ_i	κ_i if $\kappa_i = \kappa_j$ UNKNOWN if $\kappa_i \neq \kappa_j$	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

Table 5.6: Resulting size for the comparison of one dimension of λ or φ functions parameters.

only three possibilities for the lattice element: the bottom NEW, 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 φ functions is defined according to Table 5.6.

5.3.2 Propagation of Rank and Shape Information Through α Functions

Due to the possibility of dynamic growth of arrays, the analysis of α functions for the static shape inference requires a different approach. We use the lattice presented in Figure 5.4 for the analysis of α functions. In this case, the constants κ, although still considered to be in the same level in a lattice-theoretic sense, are now ordered in this level according to their values. The meet operator for α functions is defined according to Table 5.7. In this case, the range is never NEW because it must be part of the α operator.

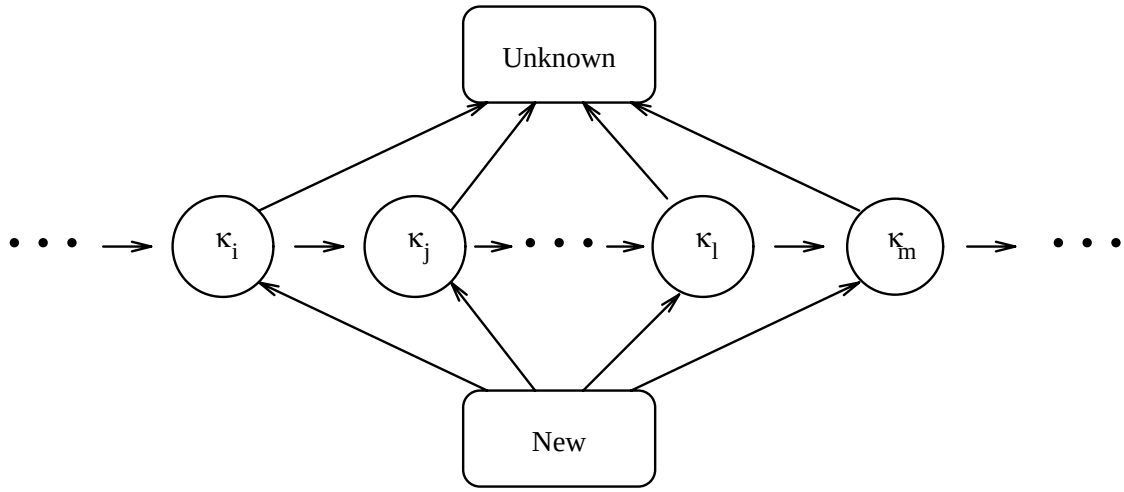

Figure 5.4: Lattice for static shape inference.

Range	Previous definition		
	NEW	κ_j	UNKNOWN
κ_i	κ_i	$\max(\kappa_i, \kappa_j)$	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

Table 5.7: Resulting size for the comparison of one dimension of α functions.

When analyzing α functions, special consideration must be taken for the cases in which there is a full use of a variable between its previous definition and an α function. In this case, due to a possible horizontal movement in the lattice, a variable may be defined with its final size that might be larger than the correct size for a particular use. Thus, the full use of the variable may result in a program error. Consider, for example, the MATLAB code segment presented in Figure 5.5(a), and its correspondent SSA representation, shown in Figure 5.5(b). Statement S1 defines a 2×2 MATRIX `B` that is multiplied by a SCALAR (`A`) in Statement S3, generating a 2×2 MATRIX `C`. Finally, Statement S4 redefines the variable `A` as a VECTOR. If there were no full use of `A` between Statements S2 and S4, there would be no problems in defining `A` as a VECTOR for the whole program. However, in this case, Statement S3 will generate a VECTOR of length 2 if `A` is considered to be a VECTOR instead of a SCALAR. One solution to this problem is to define variable `A` as a VECTOR, but to replace its full use in Statement S3 by an indexed use, as presented in Figure 5.6.

```
S1:  B = rand(2);          B1 = rand(2);
S2:  A = 10;               A1 = 10;
S3:  C = A * B;            C1 = A1 * B1
S4:  A(2) = C(2,2);        A2 = α(A1, 2, C1(2,2))
          (a)                     (b)
```

Figure 5.5: MATLAB pseudo-code segment in which there is a full use of variable between its previous definition and an α function.

```
S1:  B1 = rand(2);
S2:  A1 = 10;
S3:  C1 = A1(1) * B1
S4:  A2 = α(A1, 2, C1(2,2))
```

Figure 5.6: Indexed use to solve the array growth problem.

5.4 Functions

One of the problems for the compilation of functions is the exponential number of combinations to be considered when a function accepts several input parameters. Built-ins, however, normally accept a small number of parameters, with a limited number of intrinsic type possibilities. Thus, it is possible to generate a database for the inference process that contains all the necessary information about the built-in functions. In addition, this approach allows the possibility of backward inference, as for example with the MATLAB function `eig(A,B)`, that accept as parameters two $n \times n$ matrices to solve the generalized eigenvalue problem:

$$Ax = \lambda Bx. \tag{5.2}$$

Suppose that `A` was inferred to be NOTSCALAR (n,?) and `B` has an UNKNOWN shape. As MATLAB requires both matrices to be square matrices, we can infer that `A` and `B` are both MATRIX of shape $n \times n$.

For the case of M-files, the inlining approach was chosen over procedure cloning or interprocedural analysis, mainly due to its simple implementation. With the inliner, intrinsic

type, rank, shape, and structural information can be easily propagated inside each instance of the M-file, and the number of combinations for the properties of the input arguments is limited by the number of M-files used in the program. However, special consideration must be taken to preserve the side-effect free aspect of the M-files. Hence, for each M-file call, all local variables receive a unique name. Therefore, two instances of the same M-file are viewed independently by the compiler. We recognize a few problems with this approach. First, it generates a larger code to be compiled. Second, it may generate a much larger number of variables, especially if the same M-file is used several times with parameters having the same properties. Finally, by inlining M-files, we cannot handle recursion.

A simpler approach was taken by MathWorks in their MATLAB compiler. They compile each M-file independently, but rely on pragmas and assertions provided by the user, for the type and rank inference of the input parameters. If no information is provided by the user and type inference is not possible, they consider that all input parameters are of the same type. This approach is implemented by cloning the function for the case of REAL and COMPLEX types, and deciding during run-time with the use of a *"flag variable"* which function clone to use: the *"Complex Branch"* or the *"Real Branch"*. The downside of this approach is that if any of the input parameters has a COMPLEX type, all input parameters are considered to be COMPLEX.

5.5 Structural Inference

The main goal of the structural inference is to extract the high-level information of the MATLAB language in order to help the code generator select specific methods to perform certain operations. Hence, to perform structural inference, the compiler should use the semantic information to search for 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

```
[L, U, P] = lu(A);
y = L \ (P * b);
x = U \ y;
```

Figure 5.7: MATLAB code segment for the solution of a linear system $Ax = b$.

specialized functions for structured sparse matrices.

Consider, For example, the MATLAB code segment for the solution of a linear system $Ax = b$ using an LU decomposition, presented in Figure 5.7

The first statement calls a built-in function (`lu`) that returns a LOWER TRIANGULAR MATRIX `L`, an UPPER TRIANGULAR MATRIX `U`, and a PERMUTATION MATRIX `P`. For a regular compiler, the second statement should perform a matrix-vector multiplication (`P b`) and solve the linear system `Ly` = `Pb`. Finally, the last statement should solve the linear system `Ux` = `y`. However, by taking into consideration the semantics of the array language and knowing the properties of `L`, `U`, and `P`, the system will infer that the matrix-vector multiplication (`P * b`) is only a permutation on the vector `b`, and that the two linear systems to be solved are triangular systems. Using this information, the multiplication operation and the general linear system solve can be replaced by specialized algorithms during the code generation phase.

In several situations, MATLAB, as well as our library system, perform run-time tests to detect certain matrix structures and characteristics, as in the example described in Section 3.2.5 for the solve operation. By performing these less expensive tests, MATLAB is able to improve its performance using specialized functions. Consider, for example, the solve operation: `z = M \ b;` that appears in the preconditioned conjugate gradient (CG) [BBC+93], presented in Figure 5.8 (Statement S10). In this case, `M` is a MATRIX and `b` is a ROWVECTOR. The general method for the solution of a linear system requires $\mathcal{O}(n^3)$ operations. However, as mentioned in Section 3.2.5, if `M` were a TRIANGULAR MATRIX, it would be detected by

```
S1:  function [x, error, iter, flag] = cg(A, x, b, M, max_it, tol)
S2:  flag = 0;
S3:  iter = 0;
S4:  bnrm2 = norm( b );
S5:  if (bnrm2 == 0.0), bnrm2 = 1.0; end
S6:  r = b - A*x;
S7:  error = norm( r ) / bnrm2;
S8:  if ( error < tol ) return, end
S9:  for iter = 1:max_it
S10:   z = M \ r;
S11:   rho = (r'*z);
S12:   if ( iter > 1 )
S13:     beta = rho / rho_1;
S14:     p = z + beta*p;
S15:   else
S16:     p = z;
S17:   end
S18:   q = A*p;
S19:   alpha = rho / (p'*q);
S20:   x = x + alpha * p;
S21:   r = r - alpha * q;
S22:   error = norm( r ) / bnrm2;
S23:   if ( error <= tol ), break, end
S24:   rho_1 = rho;
S25: end
S26: if ( error > tol ) flag = 1; end
```

Figure 5.8: MATLAB function to solve the linear system Ax=b, using the Conjugate Gradient method with preconditioning.

MATLAB, and `x` would be computed in $\mathcal{O}(n^2)$ operations, using a back-substitution algorithm. Hence, the extra $\mathcal{O}(n^2)$ operation necessary to detect a TRIANGULAR structure for `M` is compensated by the cost to perform the operation using the general method for FULL MATRIX, especially since most NON-TRIANGULAR MATRICES are detected almost immediately. However, this extra overhead could be avoided if the TRIANGULAR structure of `M` were detected during compile-time. Moreover, simpler structures, such as DIAGONAL, are not detected by MATLAB, since the run-time overhead for detecting a DIAGONAL structure is as expensive as solving the problem considering `M` to be a TRIANGULAR MATRIX instead of a DIAGONAL MATRIX. If the DIAGONAL structure is detected during compile-time, however, the solve operation can be substituted by a simple $\mathcal{O}(n)$ vector division equivalent to `z = b ./ diag(M)`. This solve operation using a DIAGONAL matrix occurs normally in practice, as for example in the CG algorithm using a DIAGONAL preconditioner [BBC+93].

The following structures can be detected using the high-level semantics of MATLAB and propagated using an algebra of structures.

- VECTOR (ROW or COLUMN); from vector constructors and built-ins that reduce the rank of a MATRIX (e.g., `min` and `max` that return a VECTOR with the minimum and maximum elements of a MATRIX respectively).

- SQUARE MATRIX; as input or output of built-ins that require square matrix as parameters (e.g., `eig` for eigenvalues and eigenvectors, and `rcond` for the MATRIX conditional number estimate).

- IDENTITY; from the built-in `eye` that returns a MATRIX with ones in the diagonal and zeros elsewhere.

- PERMUTATION (only one non-zero entry in each row or column); from built-ins, such as `lu` that computes the LU decomposition, and `qr` that computes the orthogonal-triangular decomposition.

- DIAGONAL; from built-ins, such as `diag` that returns diagonal MATRICES or diagonals of a MATRIX; `svd` that computes the singular value decomposition; and `eig`.
- BANDED; from built-ins such as `diag`.
- TRI-DIAGONAL; by operating with BANDED and DIAGONAL MATRICES.
- PENTA-DIAGONAL; also by operating with BANDED, DIAGONAL MATRICES, and TRI-DIAGONAL MATRICES.
- TRIANGULAR (LOWER or UPPER); from built-ins, such as `tril` that returns a lower triangular part of a matrix; `triu` that returns an upper triangular part of a matrix; and CHOL that computes the Cholesky factorization.
- PERMUTATION of a TRIANGULAR (LOWER or UPPER); from built-ins, such as `lu`.

Another use for structural inference is to avoid the generation of dynamic code for rank inference. Consider again the code generated for the multiplication `c = a * b`, presented in Figure 3.9. If, for example, one of the matrices is known to be square, several of the tests can be avoided as both dimensions of the variable must be equal. This is shown in Figure 5.9 for the case where variable `a` is known to be a SQUARE MATRIX.

```
if (a_D1 .eq.  1) then
 if (c_D1 .ne.  b_D1 .or.  c_D2 .ne.  b_D2) then
  if ( ALLOCATED(c) ) DEALLOCATE(c)
  c_D1 = b_D1
  c_D2 = b_D2
  ALLOCATE(c(c_D1,c_D2))
 end if
 c = a(1,1) * b
else
 if (b_D2 .eq.  1) then
  if (b_D1 .eq.  1) then
   if (c_D1 .ne.  a_D1 .or.  c_D2 .ne.  a_D2) then
    if ( ALLOCATED(c) ) DEALLOCATE(c)
    c_D1 = a_D1
    c_D2 = a_D2
    ALLOCATE(c(c_D1,c_D2))
   end if
   c = a * b(1,1)
  else
   if (c_D1 .ne.  a_D1 .or.  c_D2 .ne.  1) then
    if ( ALLOCATED(c) ) DEALLOCATE(c)
    c_D1 = a_D1
    c_D2 = 1
    ALLOCATE(c(c_D1,c_D2))
   end if
   CALL DGEMV('N', a_D1, a_D2, 1.D0, a, a_D1, b,1 , 0.D0, c, 1)
  end if
 else
  if (c_D1 .ne.  a_D1 .or.  c_D2 .ne.  b_D2) then
   if ( ALLOCATED(c) ) DEALLOCATE(c)
   c_D1 = a_D1
   c_D2 = b_D2
   ALLOCATE(c(c_D1,c_D2))
  end if
  CALL DGEMM('N','N',a_D1,b_D2,a_D2,1.D0,a,a_D1,b,b _D1,0.D0,c,c_D1)
 end if
end if
```

Figure 5.9: Fortran 90 code for `c = a * b` when rank and shape of `a` and `b` are UNKNOWN, but `a` is known to be a SQUARE MATRIX.

Chapter 6

THE DYNAMIC INFERENCE MECHANISM

6.1 Dynamic Definition of Intrinsic Types

As mentioned in Section 3.2.6, every use of a variable that has UNKNOWN type requires a conditional statement to select during execution time which instance of the variable (REAL or COMPLEX) should be used. The generation of these conditional statements is straightforward, requiring only the cloning of the expression that requires this *shadow test*, and the update of the corresponding shadow variable of the output of the expression.

Variables that are initially defined as full matrices with some intrinsic type different than COMPLEX, and are later promoted to COMPLEX with the use of indexed assignments, may require special consideration. Consider, for example, the pseudo-code presented in Figure 6.1. If there were no use of the variable `A` between its initial definition (Statements S1) and the indexed assignment that promotes the array to COMPLEX (Statements S7), the static inference mechanism would detect that Statement S1 is just an array initialization and, according to the procedure described in Section 5.1.2, it would infer the intrinsic type of `A` as COMPLEX. However, since there are uses of `A` between S1 and S7, the compiler considers the variable to have dynamic type, and generates code that takes into consideration both instances of the variables, the REAL (`A_R`) and the COMPLEX (`A_C`), including a statement that copies the REAL instance of the variable into the corresponding COMPLEX instance, as

```
S1: A = ones(5);
S2: B = function(A);
S3: X = function(B)
S4: for k=2:5
S5:   C = A;
S6:   for j=2:5
S7:     B(k,j) = C(k,j) + A(k-1,j-1);
S8:     A(k,j) = sqrt(-j);
S9:   end
S10:end
```

Figure 6.1: Indexed assignment requiring intrinsic type change.

shown if Figure 6.2 (a similar analysis is valid for the variable `B` in this pseudo-code).

This approach was preferred over simply considering the array to be COMPLEX for the whole program so as to avoid the the overhead of using a COMPLEX instance of the variable before it is really necessary.

To avoid loop overhead and to maintain the correctness of the code, the copy statement should be placed outside of the outermost loop. This placement of the copy statement is easily determined. To exemplify the placement procedure, we observe the SSA representation of the program, presented in Figure 6.3. A copy statement will be necessary if the φ function inside of the outermost loop (statements P1 for `A` and P2 for `B`) has different intrinsic types for its parameters (COMPLEX for the forward parameter and another intrinsic type for the backward parameter). In this case, the corresponding copy statement is placed after the last use (that is not a λ or a φ function) of the backward definition. In this example, it is placed after Statement S2 for `A` and after Statement S3 for `B`. The last use of the backward definition is guaranteed to be outside of the loop because φ functions are placed before any other statement in the loop body.

```
S1: A_R = 1
    A_T = T_REAL
S2: B_R = function(A_R)
    B_T = T_REAL
C1: A_C = A_R
    A_T = T_COMPLEX
S3: X = function(B_R)
C2: B_C = B_R
    B_T = T_COMPLEX
S4: do k=2,5
S5:   C = A_C
S6:   do j=2,5
S7:     B_C(k,j) = C(k,j) + A_C(k-1,j-1)
S8:     A_C(k,j) = sqrt(DCMPLX(-j))
S9:   end do
S10:end do
```

Figure 6.2: Update of the COMPLEX instance of the variable and corresponding shadow value.

```
S1:  A1 = ones(5);
S2:  B1 = function(A1);
S3:  X1 = function(B1);
S4:  for k1=1:5
P1:    A5 = φ(A3,A1);
P2:    B5 = φ(B3,B1);
S5:    C1 = A5;
S6:    for j1=1:5
P3:      A4 = φ(A2,A5);
P4:      B4 = φ(B2,B5);
S7:      B2 = α(B4,(k1,j1), C1(k1,j1) + A4(k1-1,j1-1));
S8:      A2 = α(A4,(k1,j1),sqrt(-j1));
S9:    end
P5:    A3 = λ(A2,A5);
P6:    B3 = λ(B2,B5);
S10: end
```

Figure 6.3: SSA representation for the pseudo-code requiring intrinsic type change.

6.2 Dynamic Shape Inference

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 take into consideration the possibility of dynamic growth of the array. Consider, for example, the general indexed assignment: `A(i:k,j:m) = RHS`, where `A` is a REAL variable, and `i`, `j`, `k`, and `m` are INTEGER SCALARS. The dynamic code needs to check if any of the dimensions of the array `A` are going to be extended and, if so reallocate A with its new shape. Figure 6.4 presents the code generated for this indexed assignment. Notice that this dynamic code executes the procedure required by an α function.

Some optimizations in the shape inference mechanism are necessary to avoid the excessive number of tests and allocations. To illustrate this, consider the code segment of Figure 6.5, which computes the tridiagonal part of a Poisson matrix. Let us assume that the value of `n` is not known at compile time. The program resulting from a naive compilation of this code would contain allocation tests just before statements S2, S5, and S6. For example, the allocation test corresponding to statement S2 would be as shown in Figure 6.6.

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

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

```
if (k .gt.  a__D1 .or.  m .gt.  a__D2 ) then
  if (ALLOCATED(a)) then
    T0__D1 = a__D1
    T0__D2 = a__D2
    ALLOCATE(T0__R(T0__D1, T0__D2))
    T0__R = a
    DEALLOCATE(a)
    a__D1 = MAX(a__D1, k)
    a__D2 = MAX(a__D2, m)
    ALLOCATE(a(a__D1, a__D2))
    a(1:T0__D1, 1:T0__D2) = T0__R
    a(1:T0__D1, T0__D2+1:a__D2) = 0.
    a(T0__D1+1:a__D1, :)  = 0.
    DEALLOCATE(T0__R)
  else
    a__D1 = k
    a__D2 = m
    ALLOCATE(a(a__D1, a__D2))
    a = 0.
  end if
else
  if (.not.  ALLOCATED(a)) then
    a__D1 = k
    a__D2 = m
    ALLOCATE(a(a__D1, a__D2))
    a = 0.
  end if
end if
a(i:k , j:m) = RHS
```

Figure 6.4: Dynamic test for indexed assignments.

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

Figure 6.5: MATLAB code segment for the generation of a Poisson matrix.

```
    if (k .gt. P__D1 .or. k .gt. P__D2) then
      if (ALLOCATED(P)) then
        T0__D1 = P__D1
        T0__D2 = P__D2
        ALLOCATE(T0__R(T0__D1, T0__D2))
        T0__R = P
        DEALLOCATE(P)
        P__D1 = MAX(P__D1, k)
        P__D2 = MAX(P__D2, k)
        ALLOCATE(P(P__D1, P__D2))
        P(1:T0__D1, 1:T0__D2) = T0__R
        P(1:T0__D1, T0__D2+1:P__D2) = 0.
        P(T0__D1+1:P__D1, :)  = 0.
        DEALLOCATE(T0__R)
      else
        P__D1 = k
        P__D2 = k
        ALLOCATE(P(P__D1, P__D2))
        P = 0.
      end if
    else
      if (.not. ALLOCATED(P)) then
        P__D1 = k
        P__D2 = k
        ALLOCATE(P(P__D1, P__D2))
        P = 0.
      end if
    end if
S2: P(k , k) = 4
```

Figure 6.6: Fortran 90 allocation test for the MATLAB expression `P(k,k)=4`.

and the second technique is used to place the test where it will minimize the overhead.

6.2.1 Symbolic Dimension Propagation

To determine whether there is definition coverage, we use the following: *dimension propagation algorithm with symbolic capabilities.* This algorithm is similar to the range propagation algorithm used by Blume and Eigenmann for the Range Test [BE94]. Since all matrices in MATLAB have lower dimensions set to 1, our problem is simplified to determining whether the maximum value that an array index will reach is larger than the corresponding dimension in the previous assignment of the variable. If it is larger, then reallocating the array is necessary; otherwise, no dynamic test is necessary for the assignment being considered.

Our dimension propagation algorithm symbolically computes the maximum value of each scalar subscript expression that can be used to generate `ALLOCATE` statements. This information is obtained by tracing the indices of the array to their earliest definitions in the AST representation of the program, following an on-demand approach similar to that introduced in [TP95]. In some situations (such as non-scalar indices or indirections), this symbolic inference is unable to determine the maximum value. In this case, the compiler sets the corresponding information as UNKNOWN, and dynamic allocation is required.

Algorithm: *Symbolic Dimension Propagation.*

Input: An AST representation of a MATLAB program.

Output: The symbolic value for each SCALAR variable, and the symbolic dimensions for the dynamically allocated variables.

This algorithm is implemented as a single compiler pass that traverses the AST in lexicographic order gathering symbolic information. Its goal is to synthesize the following information:

1. For SCALAR assignments (e.g., `S = RHS`):

- A pointer (P_s) to an AST node corresponding to the definition of the RHS.
- A list (L_s) containing symbolic expressions that define the value of a SCALAR. Each node in the list contains a pointer to a variable definition (V) in the AST, and a constant value indicating the numeric difference between (V) and the variable being assigned. If the constant value is zero, the two variables are symbolically the same.

2. For definitions of dynamic variables (two-dimensional allocatable arrays):
   - Two pointers (P_nr and P_nc) to AST nodes, corresponding to the SCALAR variables that define respectively the number of rows and number of columns of the array.
   - Two lists (L_nr and L_nc) similar to L_s, respectively for the number of rows and number of columns of the array.

The algorithm could be rewritten using a node in the list L_ with a value equal to zero to replace the corresponding pointer P_. However, the pointers are used to speedup the search process.

To illustrate the procedure for symbolic dimension propagation, we consider the SSA representation of an extension of the code that generates the tridiagonal part of a Poisson matrix, as presented in Figure 6.7. The action taken by the algorithm depends on the kind of node being visited, as described next:

Scalar assignments :

- If the RHS is a SCALAR (e.g., S1 and S2): The pointer P_s is updated with the definition of the variable in the RHS. The algorithm traces back to the earliest definition, using the pointers P_s already updated. So, in Statement S2, P_s will point to the definition of m_1 in S0 instead of the definition of x_1 in S1. This is possible because the pointer P_s for

```
S0:  load %(m_1)
S1:  x_1 = m_1;
S2:  n_1 = x_1;
S3:  P_5 = NULL
S4:  for k_1=1:n_1
S5:    P_6 = φ(P_1, P_5);
S6:    P_1 = α(P_6,(k_1,k_1),4);
S7:  end
S8:  P_4 = λ(P_1, P_5);
S9:  T__1_1 = n_1-1;
S10: for j_1=1:T__1_1
S11:   P_8 = φ(P_3, P_4);
S12:   T__2_1 = j_1+1;
S13:   P_2 = α(P_8,(j_1,T__2_1),-1);
S14:   T__3_1 = j_1+1;
S15:   P_3 = α(P_8,(T__3_1,j_1),-1);
S16: end
S17: P_7 = λ(P_3, P_4);
S18: Z_1 = P_7
```

Figure 6.7: Extension of a MATLAB code to generate a Poisson matrix.

x_1 will be already pointing to m_1; therefore, the algorithm can backtrack using this information.

- If the RHS is a SCALAR expression (e.g., S9, S12, and S14): The symbolic expression is added to the list L_s of the corresponding variables. Backtracking is also used for this list. So, for example, in Statement S9, the information that $T\_\_1_1$ is one less than m_1 is saved. For the case of $T\_\_2_1$ and $T\_\_3_1$, both fields can be updated. First, L_s is updated with the information that $T\_\_2_1$ is one greater than j_1. Second, as P_s from j_1 will indicate the maximum value that j_1 will reach is $T\_\_1_1$, with simple symbolic computation m_1 can be assigned to the pointer P_s of $T\_\_2_1$. A similar update is performed for $T\_\_3_1$.
- If the RHS is a triplet expression (e.g., `LHS = start:stride:end`): The symbolic information is obtained according to the procedure above. It considers the symbols `start` or `end` of the vector triplet as the RHS, depending on the sign of `stride`. If `stride` is positive (or not present in the triplet expression), the symbolic information is obtained from `end`. Otherwise, it is derived from `start`.

Full array assignments :

- If the RHS is a built-in that creates arrays (e.g., `zeros`, `rand`, `eye`): The symbolic information (P_nr, P_nc, L_nr, and L_nc) is updated according to the parameters of the built-in function by backtracking to the earliest definition using the pointers P_s from the built-in parameters.
- If the RHS is an array (e.g., S18): The symbolic information is updated according to the symbolic information from the RHS.
- If the RHS is a built-in function that returns matrices with the same shape as the input parameter (e.g., `sin`, `sqrt`, `abs`): The symbolic

information is updated according to the symbolic information of the built-in parameter.

- If the RHS is a two-dimensional conformable expression (e.g., `C=A+B`): For each dimension, if the dimension of both operands can be proven to be symbolically equal, the symbolic pointer for `C` is updated with the corresponding information of one of the operands. The symbolic list is updated with the union of the lists of the two operands.
- If the RHS is a one-dimensional conformable expression (e.g., `C=A*B`): In this case, the symbolic information for `C` cannot be inferred, unless some information about the values of the conformable dimension of the operands is known. Consider, for example, the multiplication `C = A * B`, where `A` and `B` are `n`$\times$`q` and `p`$\times$`m`, respectively. We cannot infer that `C` will be an `n`$\times$`m` matrix without knowing the actual values of `n`, `q`, `p`, and `m`. If `n` and `q` are both equal to 1, then `C` will be a `p`$\times$`m` matrix. Similarly, if `p` and `m` are both equal to 1, then `C` will be an `n`$\times$`q` matrix. Thus, without further information about `n`, `q`, `p`, and `m`, it is impossible to infer the symbolic information for the variable `C`. In this case, if the conformable dimensions `q` and `p` are known to be different than 1, then P_nr will be set to `n`, P_nc will be set to `m`, and the lists L_nr and L_nc will be updated according to the left and right operands, respectively.

Indexed assignments (α functions) (e.g., S6, S13, and S15) :

- For each dimension, compare the symbolic information from the index (P_s and L_s) with the symbolic information from the previous definition (P_nr and L_nr, or P_nc and L_nc). The symbolic information for the indexed assignment is the largest of the two. Consider for example the first dimension in S15. The index $\texttt{T\_\_3}_1$ has P_s set to $\texttt{m}_1$, and $\texttt{P}_8$ has

P_nr also set to $\mathtt{m}_1$. Therefore, P_nr for $\mathtt{P}_3$ will be set to $\mathtt{m}_1$, and L_nr will have the union of the two lists: L_s from $\mathtt{T\_\_3}_1$ and L_nr from $\mathtt{P}_8$. If P_s from $\mathtt{T\_\_3}_1$ and P_nr from $\mathtt{P}_8$ were different, then symbolic computation would be performed to define which value is larger.

λ **functions** :

- The symbolic information from both parameters are compared. If they have different information, the output is set as UNKNOWN.

φ **functions** :

- An iterative process similar to the one described in Section 5.1.1 for the algorithm for intrinsic type inference of φ functions is performed. The symbolic information of the output of the φ function is assigned *tentatively* according to the backward parameter. After the symbolic information for the forward parameter is computed, the symbolic information of the φ function is reevaluated as a λ function. If the output is UNKNOWN, then all information previously defined based on the initial choice is nullified.

End *algorithm for symbolic dimension propagation.*

6.2.2 Coverage Analysis

To solve the coverage problem for an indexed array assignment, we compare the shape resulting from the indexed assignment with the shape of the array before the assignment, using the information from the symbolic dimension propagation algorithm. 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.

Consider again the code segment presented in Figure 6.7. From the scalar assignment in S4, the compiler determines that the maximum value of the variable $\mathtt{k}_1$ will be $\mathtt{n}_1$. Using this

information, the compiler determines that S6 creates a matrix P with shape $n_1 \times n_1$. From S10, the compiler determines that the maximum value of j_1 will be $T\_\_1_1$, that is equal to n_1-1. Notice that by using simple symbolic algebra capabilities it is possible to determine that $T\_\_2_1$ and $T\_\_3_1$ are equal to j_1+1 and $\leq n_1$. Consequently, the shape of P is not expanded in S13 or S15; therefore, it is not necessary to generate dynamic allocation tests for these statements.

6.2.3 Placement of Dynamic Allocation

If dynamic allocation is necessary, the following algorithm for placement of dynamic allocation is activated:

Algorithm: *Placement of Dynamic Allocation.*

Input: An α function that requires dynamic allocation.

Output: Location in the AST for the dynamic allocation.

1. Using the pointer P_s described in the algorithm for symbolic dimension propagation, trace the the definition of the variable that determines the symbolic dimension (S).
2. Trace the SSA representation from a variable assignment to its previous full definition (D). If there is no previous definition for the assigned variable, the `ALLOCATE` can be placed at any point between S and the assignment.
3. If an allocation test is required after D, it can be placed after both S and the last use of D (that is not a λ or a φ function).

End *algorithm for Placement of Dynamic Allocation.*

If possible, the dynamic allocation should be placed outside of the outermost loop. As mentioned before, due to the algorithm for the generation of φ functions, the last use of D is guaranteed to be outside of the loop. Hence, if S is outside of the outermost loop, the

`ALLOCATE` statement can be placed outside of the outermost loop, thus avoiding the loop overhead.

Chapter 7

EXPERIMENTAL RESULTS

This chapter presents the experiments performed to measure both the effectiveness of the internal phases of the inference mechanism and the overall effectiveness of the compiler. Three sets of experiments were performed. In the first set, the performance of compiled codes were compared with their interpreted MATLAB execution, with the C-MEX files compiled by the MathWorks compiler, and with Fortran 90 hand-written versions of the same algorithms. In the second set, the importance of each of the major phases of the inference system was measured. Finally, in the third set, a scalability study was performed to determine how problem size affects the relative speed of the programs.

The following sections present a description of the test programs and the computational environment where the experiments were performed, the evaluation of the overall effectiveness of the compiler, the evaluation of the inference phases, and the scalability analysis.

7.1 Description of the Test Programs

For the first set of experiments, we ran 12 MATLAB programs on a single processor of an SGI Power Challenge and on a Sun SPARCstation 10. 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). These parameters were chosen in order to execute each MATLAB program in about one minute on a Sun SPARCstation 10.

Test Programs:		Problem size	Lines	Source
Preconditioned Conjugate Gradient method	(CG)	$420 \times 420^*$	36	a
Successive Overrelaxation method	(SOR)	$420 \times 420^*$	29	a
Quasi-Minimal Residual method	(QMR)	$420 \times 420^*$	91	a
Adaptive Quadrature Using Simpson's Rule	(AQ)	1 Dim. (7)	87	b
Crank-Nicholson solution to the heat equation	(CN)	321×321	29	b
Finite Difference solution to the wave equation	(FD)	451×451	28	b
Dirichlet solution to Laplace's equation	(Di)	41×41	39	b
Galerkin method to solve the Poisson equation	(Ga)	40×40	48	c
Two body problem using 4th order Runge-Kutta	(RK)	3200 steps	66	c
Two body problem using Euler-Cromer method	(EC)	6240 steps	26	c
Incomplete Cholesky Factorization	(IC)	400×400	33	d
Generation of a 3D-Surface	(3D)	$51 \times 31 \times 21$	28	d

Source:
a: [BBC+93] b: [Mat92c] c: [Gar94] d: Colleagues

* A stiffness matrix from the Harwell-Boeing Test Set (BCSSTK06) was used as input data for these programs.

Table 7.1: Test programs.

A brief description of the test programs is presented in Table 7.1. These programs can be classified into different groups, depending upon certain characteristics of the MATLAB code and its execution. The main characteristics are: use of indexed assignments; requirement of array growth during execution time; execution time dominated by built-in functions; and use of multi-typed built-ins. We refer to the programs that have execution time dominated by build-in functions as *library-intensive programs*[1], while loop-based programs that perform mostly elementary scalar operations (using SCALARS or element-wise access of VECTORS or MATRICES) are referred to as *elementary-operation intensive*. Finally, a program that concentrates its execution time with memory allocations and data movements is referred to as a *memory-intensive program*.

A description of the main characteristics of each program is presented next. Notice that all programs, with the exception of CG, SOR, and QMR, use indexed assignment.

[1]Notice that in our analysis, operations such as matrix-matrix multiplication and matrix-vector multiplication are also considered built-ins due to their use of specialized library functions to perform the computation.

CG: is an iterative method for the solution of linear systems. It is an implementation of the preconditioned conjugate gradient algorithm using diagonal preconditioner. This is a library-intensive program due to the computation of matrix-vector multiplications. It has no indexed assignments.

SOR: is also an iterative solver. Its is a library-intensive program due to a triangular matrix-vector multiplication and a triangular solve. This program has no indexed assignments.

QMR: is another iterative solver with no indexed assignments. Its execution time is also dominated by matrix-vector multiplications. In addition, it uses a multi-typed function (`sqrt`) to compute the expression:

$$\gamma = \frac{1}{\sqrt{1+\theta^2}}. \tag{7.1}$$

γ is used to update a VECTOR containing the residual and a VECTOR containing the approximation. These VECTOR updates, however, require only $\mathcal{O}(n)$ computations, and are not propagated through the part of the code that executes the matrix-vector multiplications (which require $\mathcal{O}(n^2)$ computations).

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

$$\int_{-1}^{6} 13 * (x - x^2) e^{\frac{-3x}{2}} dx. \tag{7.2}$$

This is a memory-intensive program because the adaptive quadrature method adjusts the integration interval to smaller subintervals, when some portions of the curve have large functional variation [Mat92c]. This refinement in the integration process requires data movements and dynamic reallocation of an array as new subintervals are generated.

CN: is a numeric approximation method for the solution of parabolic differential equations (the heat equation). This is an elementary-operation intensive program that performs indexed updates to a two-dimensional grid.

FD: is a numeric approximation method for the solution of hyperbolic differential equations (the wave equation). This is also an elementary-operation intensive program that performs indexed updates to a two-dimensional grid.

Di: is an iterative method for the solution to Laplace's equation. This is another elementary-operation intensive program requiring element-wise access of grid elements. Another of its characteristics is the use of a multi-typed built-in (`sqrt`) to compute Expression 5.1, presented in Section 5.2. As mentioned before, in this program, the output of the square root (ω) is always a REAL value. Thus, the penalty for not being able to infer the intrinsic type of ω would be very large, because it is propagated through the grid used by the program.

Ga: is a numeric approximation method to solve the Poisson equation in two-dimensions using the Galerkin method. This is also an elementary-operation intensive program. This test program computes the charge distribution for a two-dimensional dipole using a multi-typed built-in (`log`). The logarithmic function is used for the computation of the expression:

$$\Phi(r) = \frac{-\lambda}{2\phi\varepsilon_0}\left\{ln\left[\left|r - \left(r_c + \frac{1}{2}d\right)\right|\right] - ln\left[\left|r - \left(r_c + \frac{1}{2}d\right)\right|\right]\right\} \tag{7.3}$$

This expression is used only to save the partial result for plotting purposes. Thus, its output is not propagated through the program. Although our value-propagation algorithm is able to detect that the output of the logarithm function is REAL, the penalty of not inferring it would be very small.

EC: is an ordinary differential equation problem. It computes the orbit of a comet about the Sun using the Euler-Cromer method. Its main characteristic is the utilization of the built-in `norm` to compute norms of VECTORS of size 2. Also, due to the lack of "pre-allocation" in the program[2], the VECTORS that are used to save the results grow

[2] A common practice of MATLAB programmers is to pre-allocate VECTORS and MATRICES using built-ins, such as `zeros`, to avoid allocation overhead inside of loops.

during the MATLAB execution.

RK: uses a different approach to solve the same problem as EC. In this case, it uses the Runge-Kutta method. It also performs computations of built-ins using VECTORS of size 2, and requires reallocations of VECTORS during the MATLAB execution.

IC: uses a double-nested loop to compute the incomplete Cholesky factorization of a matrix. Its main characteristic is the use of a multi-typed built-in (`sqrt`). In this case, as shown in the program segment presented in Figure 7.1, the square root may result in a COMPLEX output. However, due to the conditional statement following the square root[3], the COMPLEX value of `r` is never assigned to `L`. Thus, the program does not need to perform COMPLEX arithmetic. Our inference mechanism is not yet able to detect that the COMPLEX result is not used and, due to the promotion of intrinsic types described in Section 5.1.2, the compiled program cannot avoid the use of COMPLEX variables in this case.

3D: uses a three-nested loop to generate the 3-dimensional surface. This is a library-intensive program due to the computation of eigenvalues with the use of the built-in `eig`.

7.2 Evaluation of the Overall Compiler Effectiveness

Tables 7.2 and 7.3 present the execution times in seconds for all programs in our test set running on the SGI Power Challenge and on the Sun SPARCstation 10, respectively. For each individual test the time presented in the table is the best time out of five runs. The Fortran 90 programs were compiled on the SGI using their native Fortran 90 compiler with the optimization flag "O3". Due to the lack of a native Fortran 90 compiler on our SPARCstation,

[3]In MATLAB, a LOGICAL expression can operate with COMPLEX operands; however, it considers only the REAL part. Hence, if the square root function returns a COMPLEX value, its REAL part is zero. Thus, the logical test `(sqrt(...) <= 0)` will always return `true` if the output of the square root is COMPLEX.

```
for j = 1:n
...
  r = sqrt(L(j,j) - s);
  if (r <= 0)
    Error = j;
    L(j,j) = 1;
  else
    L(j,j) = r;
  end
  ...
end
```

Figure 7.1: MATLAB code segment for the Incomplete Cholesky Factorization (IC).

the Fortran 90 programs were first translated to Fortran 77 with the use of VAST 90 [Pac93], and then compiled with the Sun Fortran 77 compiler using the optimization flag "O3". The C-MEX-files generated by the MathWorks MATLAB to C Compiler (MCC) were compiled on the SPARCstation with the GNU C compiler using the optimization flag "O3". On the SGI, they were compiled both with the SGI native C compiler using the highest possible 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 was loaded using interpreted MATLAB commands (not timed) and provided to the MEX-files as function parameters. Assertions indicating the intrinsic type and rank of the loaded variables were added to the M-files to provide MCC with the same information that was extracted by our compiler from the loaded variables (described in Section 3.2.5).

Figures 7.2 and 7.3 present the speedups of the compiled codes over the interpreted MATLAB execution running on the SGI Power Challenge and on the Sun SPARCstation 10, respectively. In both figures, the darker bars represent the speedup of our compiler over MATLAB, while the lighter bars represent the speedup of MCC over MATLAB. Due to the large difference in performance for some of the programs, the speedups are shown in

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

Table 7.2: Execution times (in seconds) running on an SGI Power Challenge.

Program	MATLAB	MCC	FALCON	Hand coded
AQ	59.53	25.82	25.71	15.11
CG	46.55	58.47	20.68	19.15
CN	60.98	1.85	0.36	0.34
Di	75.28	4.43	0.26	0.26
FD	49.02	1.30	0.28	0.28
Ga	54.87	1.32	0.70	0.69
IC	53.92	5.12	1.97	0.61
3D	64.73	30.77	10.77	10.63
EC	72.37	17.33	0.19	0.16
RK	53.45	19.22	0.23	0.18
QMR	54.93	60.33	13.88	12.80
SOR	62.05	66.38	24.43	7.64

Table 7.3: Execution times (in seconds) running on a Sun SPARCstation 10.

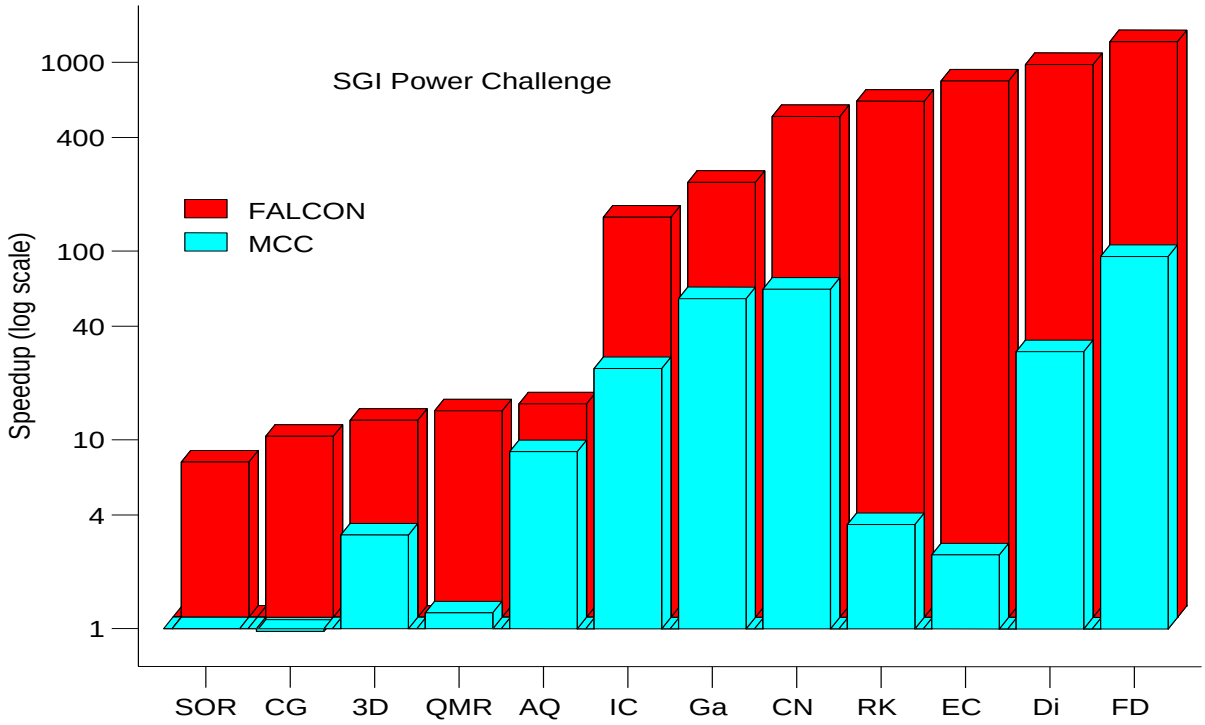

Figure 7.2: Speedup of compiled programs over MATLAB, running on the SGI Power Challenge.

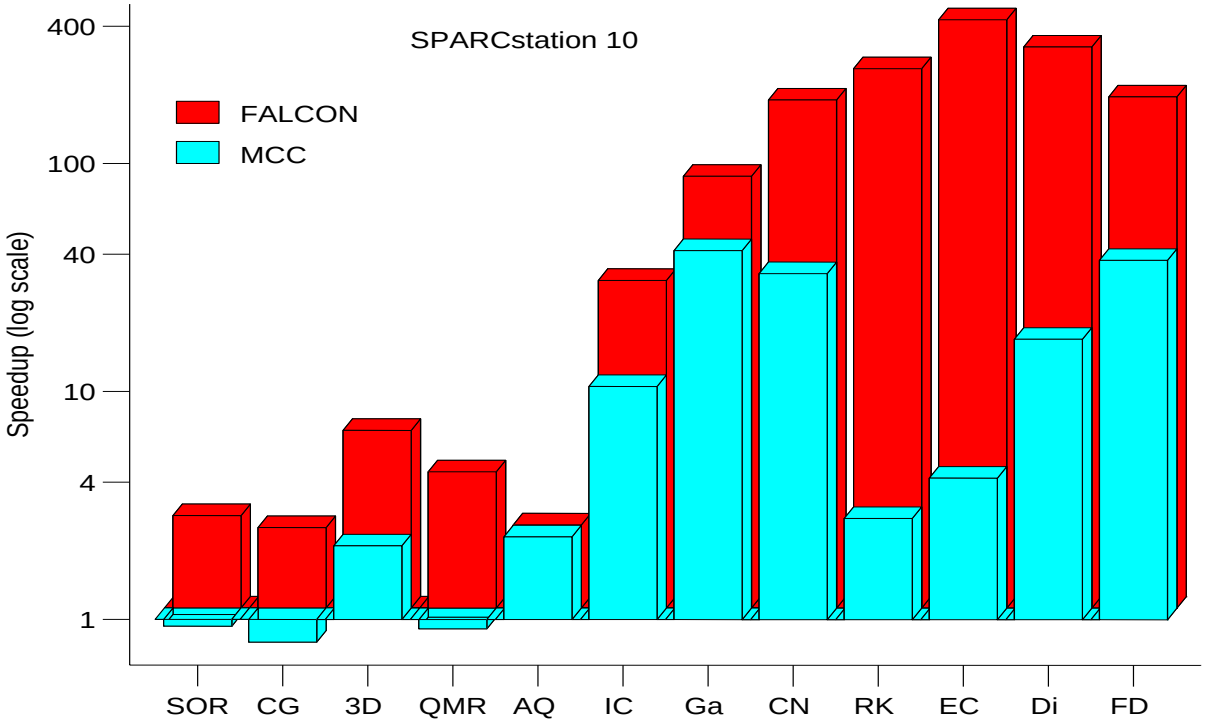

Figure 7.3: Speedup of compiled programs over MATLAB, running on the Sun SPARCstation 10.

logarithmic scale. We observe that the performance of the Fortran 90 programs on the SPARCstation 10 is partly affected by the use of VAST 90. As mentioned before, VAST 90 translates the Fortran 90 into Fortran 77, causing a performance degradation on the final code.

7.2.1 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 is better than the respective interpreted execution, and the range of speedups is heavily dependent on the characteristics of the MATLAB program.

Library-intensive programs (e.g., CG, SOR, and 3D) have a small speedup compared to MATLAB. In these programs, the speedup obtained is primarily attributed to the overhead of interpretation since, in general, the built-ins use the same optimized library functions that are called by our compiler and by the MathWorks compiler.

The speedup obtained by AQ resulted from the better handling of indexed assignments and the reallocation of matrices by the compiled program. However, according to the scalability study to determine how problem size affects the relative speed of the programs, discussed in Section 7.4, this improvement varies considerably, depending upon the number of reallocations required by the program, which is in turn dependent upon the input data set and the function being used for the numerical integration.

Finally, elementary-operation intensive programs 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 the larger overhead of the indexed assignments within the interpreted code.

7.2.2 Comparison of Compiler Generated Programs with the Hand-written Fortran 90 Programs

When comparing the hand-coded Fortran 90 programs with the compiler generated versions, we observe that, for most programs, the performance of the compiled versions is very close to the performance of the hand-written programs.

The largest performance differences occur with IC and SOR. In both cases, the hand-written code performed more than four times faster on the SGI, and more than three times faster on the SPARCstation. The reason for the performance difference with the IC program was the inability of the inference mechanism to detect that the conditional statement shown in Figure 7.1 would prevent the array `L` to became COMPLEX. Hence, the compiled code performs COMPLEX arithmetic for most of the program, while the optimized program uses REAL variables for the same operations.

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

```
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 this information 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, the price of detecting the triangular structure of the matrix is that it requires $\mathcal{O}(n^2)$ operations. Moreover, in both cases, the matrix multiplication is performed using a generalized function for full matrices that performs $2n^2$ operations, while 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. We observe that with the

implementation of the structural inference mechanism, discussed in Section 5.5, the compiler will be able to detect the triangular structure of the matrices, and the performance of the generated code will be closer to the hand-optimized code.

Other performance differences that are worth mentioning are from EC, RK, and AQ. In the first two programs, the difference in performance is attributed to the computation of the `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, while the hand-optimized programs perform the computation using a single expression which takes into consideration that the vector has only two elements. Moreover, due to the granularity of the programs, the performance difference becomes more noticeable when the programs are executed on the SGI.

Finally, the performance difference observed with AQ is primarily a result of the code generated by the compiler for the reallocation of matrices, as presented in Figure 6.6. In the hand-written code, because of a better knowledge of the algorithm, it is possible to avoid part of the copy of the old values to the expanded matrix and the initialization of the expanded part to zero.

7.2.3 Comparison with the MathWorks MATLAB Compiler

Figures 7.4 and 7.5 present the speedups of the codes generated by our compiler over MCC's codes. With the exception of AQ on the SPARCstation 10 which, as described previously, spends most of its time performing reallocations and data movements, all other codes generated by our compiler ran faster than their MCC counterparts. We observe from Figure 7.3 that in three cases (CG, SOR, and QMR) MCC generated programs that ran slower than MATLAB on the SPARCstation 10.

Three programs generated by our compiler (RK, EC, and Di) had significantly better performance than the corresponding MCC versions. The primary reason for these differences, is lack of more in depth inference analyses by the MathWorks compiler. The MathWorks

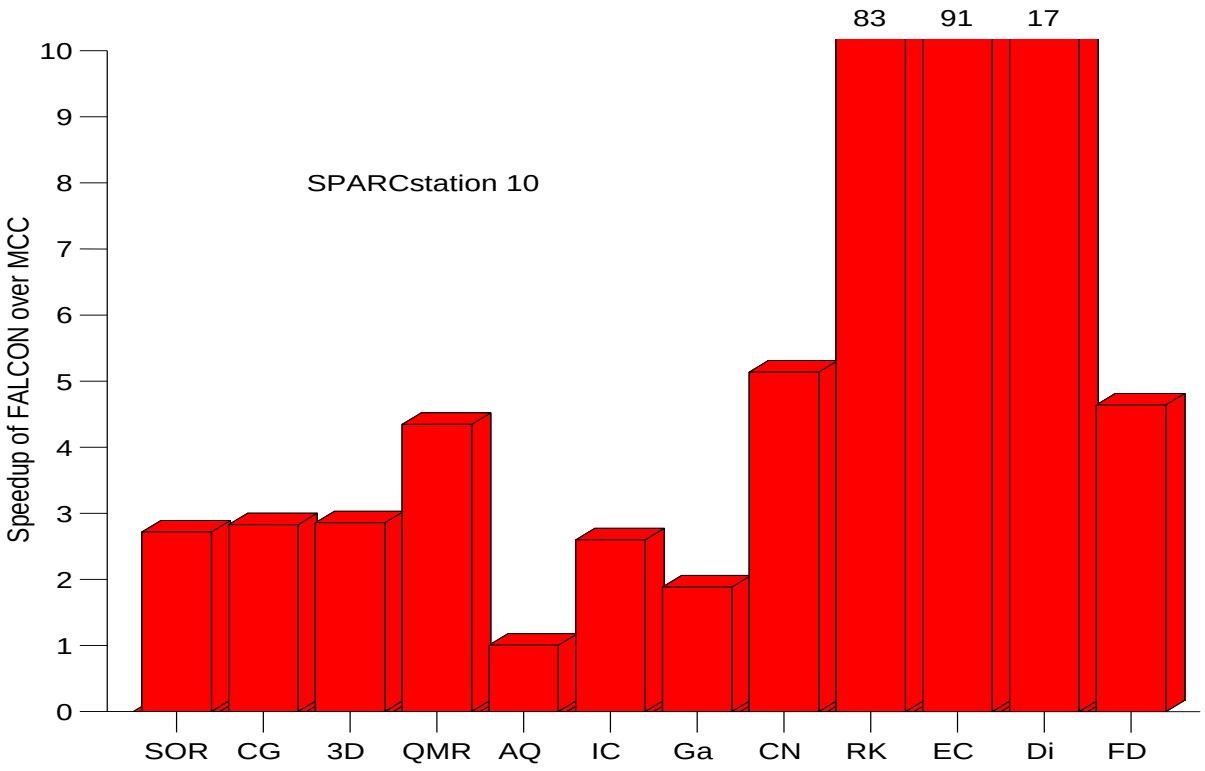

Figure 7.4: Speedup of FALCON's compiler over MCC on a SPARCstation 10.

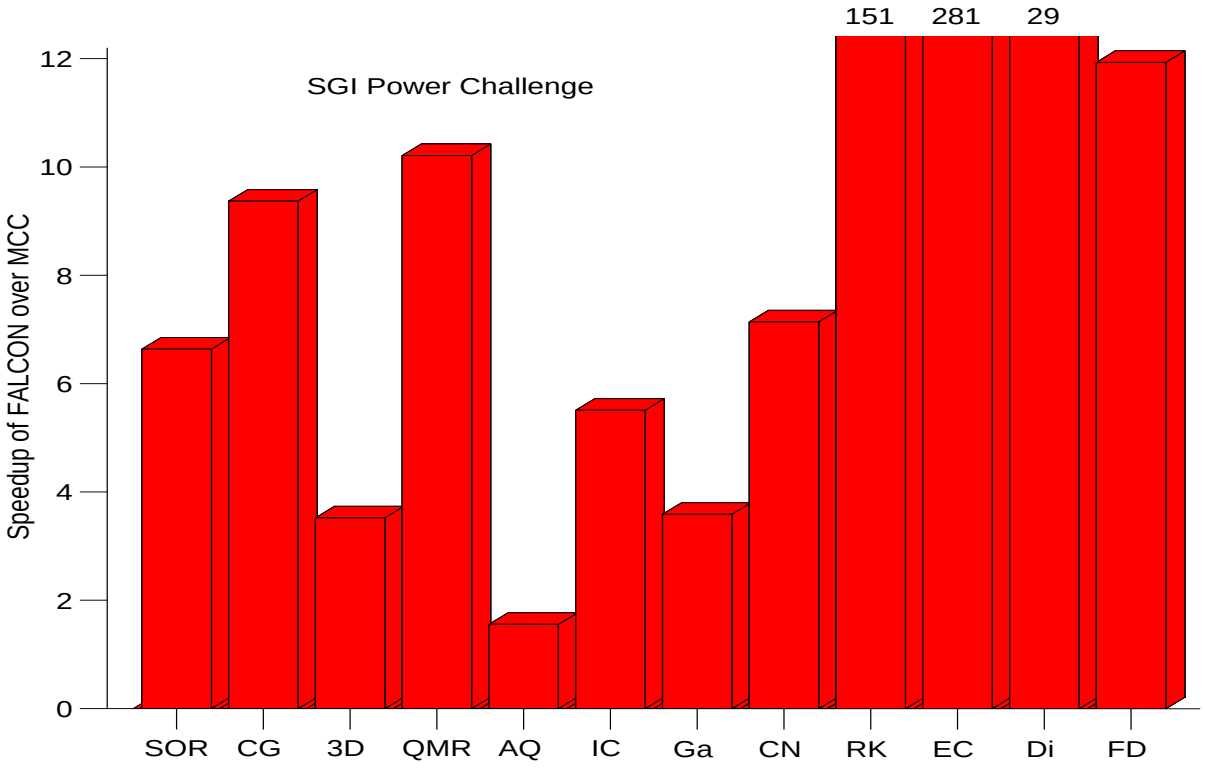

Figure 7.5: Speedup of FALCON's compiler over MCC on an SGI Power Challenge.

compiler does not perform use-coverage analysis and, as mentioned before, simplifies the handling of multi-typed built-ins by considering that they always return COMPLEX output. Moreover, as described in [Mat95], 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 "*callback 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 callback functions generate an overhead that, in this case, is not amortized due to the size of the vectors. Moreover, MCC does not appear to infer the rank VECTOR, and instead treats both VECTORS and MATRICES with the same data structure. Small vector size generates another overhead problem for MCC. Furthermore, the lack of pre-allocation of variables in the MATLAB code is also responsible for the degradation of the performance of these MCC codes. Our compiler, by contrast, is able to allocate all matrices in these programs outside the main loop, because of our symbolic propagation analysis.

Finally, the better performance from Di results from our value-propagation analysis. In this case, the intrinsic type inference mechanism can determine that the Expression 5.1 will always return a REAL value, whereas MCC assumes the output of multi-typed built-in to be always of type COMPLEX. Thus, the code generated by MCC for Di uses COMPLEX variables for most of its computations, whereas ours uses only REAL variables.

7.3 Evaluation of the Inference Phases

For the analysis of the major phases of the compiler and for the scalability analysis, we run a select subset of the programs above, with each program representing a different characteristic of the MATLAB codes. For the evaluation of the inference phases, the following programs

were selected:

- AQ, due to its reallocation of arrays;
- CG, representing the group of programs without indexed assignments;
- 3D, as a library-intensive program;
- Di, due to its use of a multi-typed built-in; and
- FD, representing the elementary-operation intensive programs.

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

No inference: all variables were declared as two-dimensional allocatable COMPLEX arrays.

Only shape and rank inference: all variables were of type COMPLEX.

No intrinsic type inference: shape, rank, and symbolic inference were activated.

Only intrinsic type inference: all variables declared as two-dimensional allocatable arrays.

No symbolic inference: type, shape, and rank inference were activated.

Complete inference: all inference phases were performed.

Figure 7.6 shows a graphical comparison of the execution times for all programs, using all six combinations. The execution times in seconds are also presented in Table 7.4.

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

Shape and rank inference had a large influence on performance for all other programs, with improvements ranging from 3 times faster for CG to almost 30 times faster for Di. The main reason for this improvement is the reduction in overhead for dynamic shape inference, especially for scalars. When dynamic shape inference is necessary for a matrix, the overhead generated by the compiler may be amortized by the subsequent operation and assignment to the matrix, depending on its size and the number of floating point operations performed by the expression. On the other hand, a typical scalar assignment 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 the computational intensive programs (CG, Di, and FD). In these cases, when shape inference and dimension propagation were activated, the speedup resulting from type inference ranged from 3.8 to 5. On the other hand, type inference had very little effect on AQ since it spends most of its time performing data movements.

inference phases	AQ	CG	3D	Di	FD
no inference	27.93	3.49	3.92	2.45	2.93
only shape and rank	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 7.4: Execution times in seconds when inference phases were deactivated.

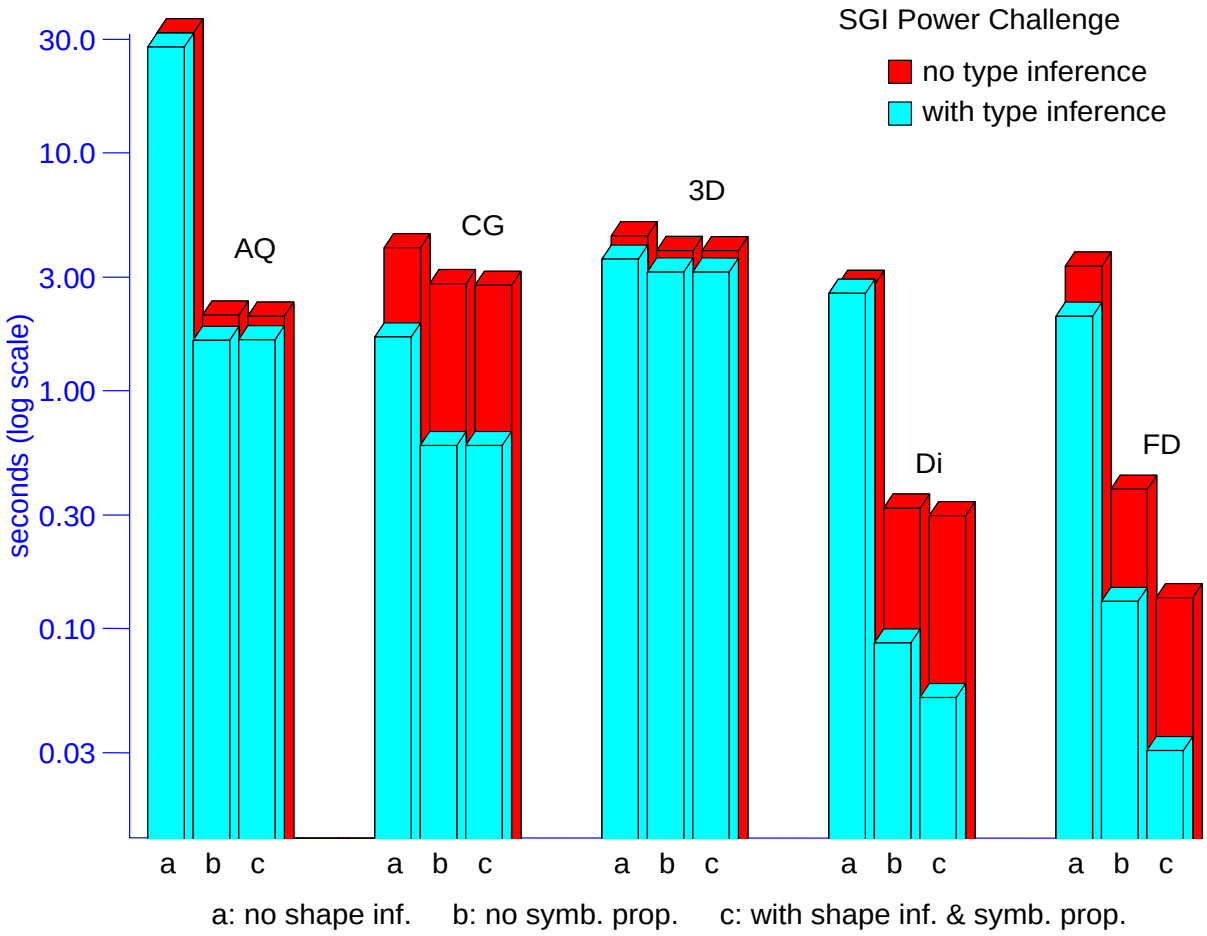

Figure 7.6: Comparison of the inference phases.

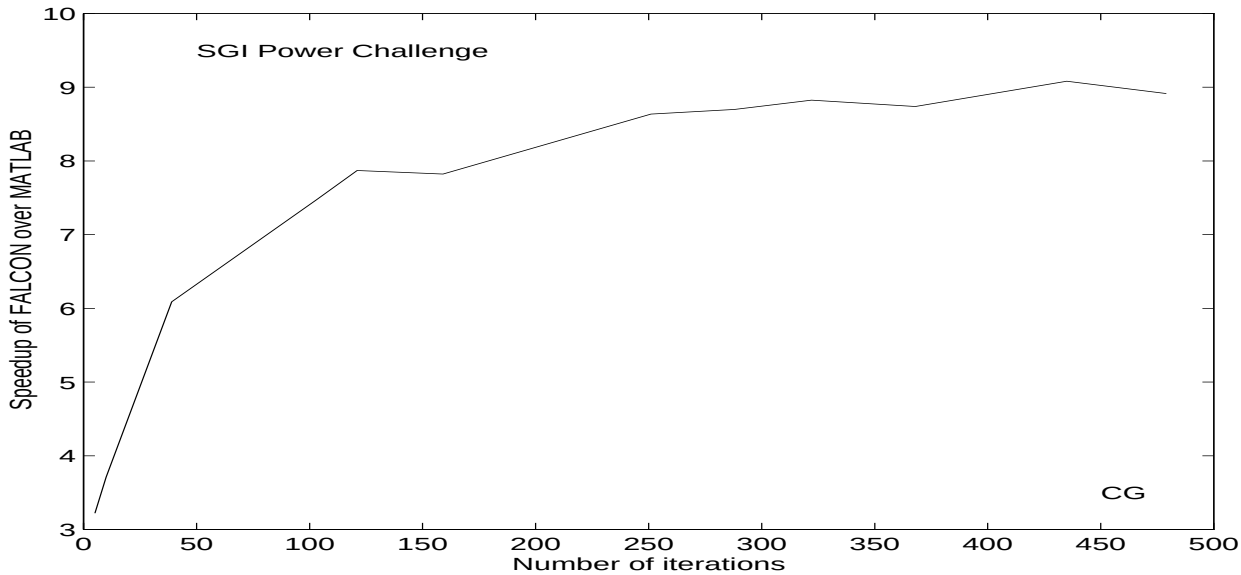

Figure 7.7: CG speedups on the SGI Power Challenge when increasing the number of iterations.

7.4 Scalability Analysis

In our scalability analysis, we studied the behavior of three programs with respect to changes in problem size (CG and FD), number of iterations (CG and AQ), and memory requirements (AQ). CG was selected because it is a library-intensive program. FD represents the elementary-operation intensive programs. And, finally, AQ was selected due to its memory-intensive characteristics. Unfortunately, these times had to be obtained while running in multi-user mode.

7.4.1 Analysis of Library-intensive Programs

For this study, we performed two sets of runs using the program CG. In the first set of experiments, we measure the speedup of the compiled Fortran 90 programs over MATLAB on the SGI Power Challenge, varying the tolerance from 10^{-1} to 10^{-15} with fixed problem size (using the same 420×420 matrix stiffness matrix). The corresponding speedup curve is presented in Figure 7.7.

When increasing the number of iterations for a fixed problem size, the computation time

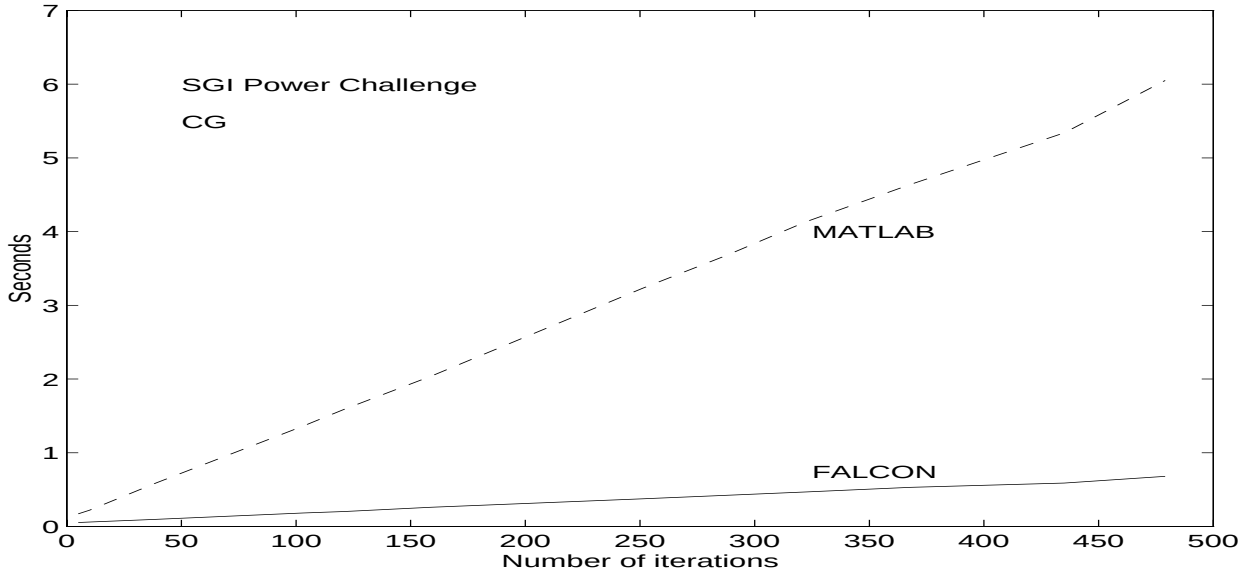

Figure 7.8: CG Execution time on the SGI Power Challenge with fixed problem size.

increases linearly, as shown in Figure 7.8, which presents the time in seconds for both the MATLAB execution and the compiled execution. Hence, the speedup curve is expected to reach an asymptotic value for a large number of iterations. This value appears to be around 9 for the problem size used. When only a small number of iterations is executed, the speedup is expected to be small because the initial overhead of the program has yet to be amortized.

In the second experiment, we measured the speedup of the compiled Fortran 90 code over MATLAB on the SGI Power Challenge using a fixed number of iterations and varying the problem size. For this experiment, we used a larger matrix from the Harwell-Boeing Test Set (BCSSTK10). This matrix has 1086×1086 elements. Several runs were executed, each one using a subset of the matrix with different size. In all runs, the number of iterations was fixed to 145 and the tolerance was set to 10^{-15}. The corresponding speedup curve is presented in Figure 7.9, and the Mflop ratios for the compiled program and the interpreted execution are presented in Figure 7.10.

The speedup curve presents four regions that require observations. First, when running with small problem sizes we observe a higher speedup because the time spent by MATLAB to perform the built-in functions is not enough to amortize the overhead of the rest of the

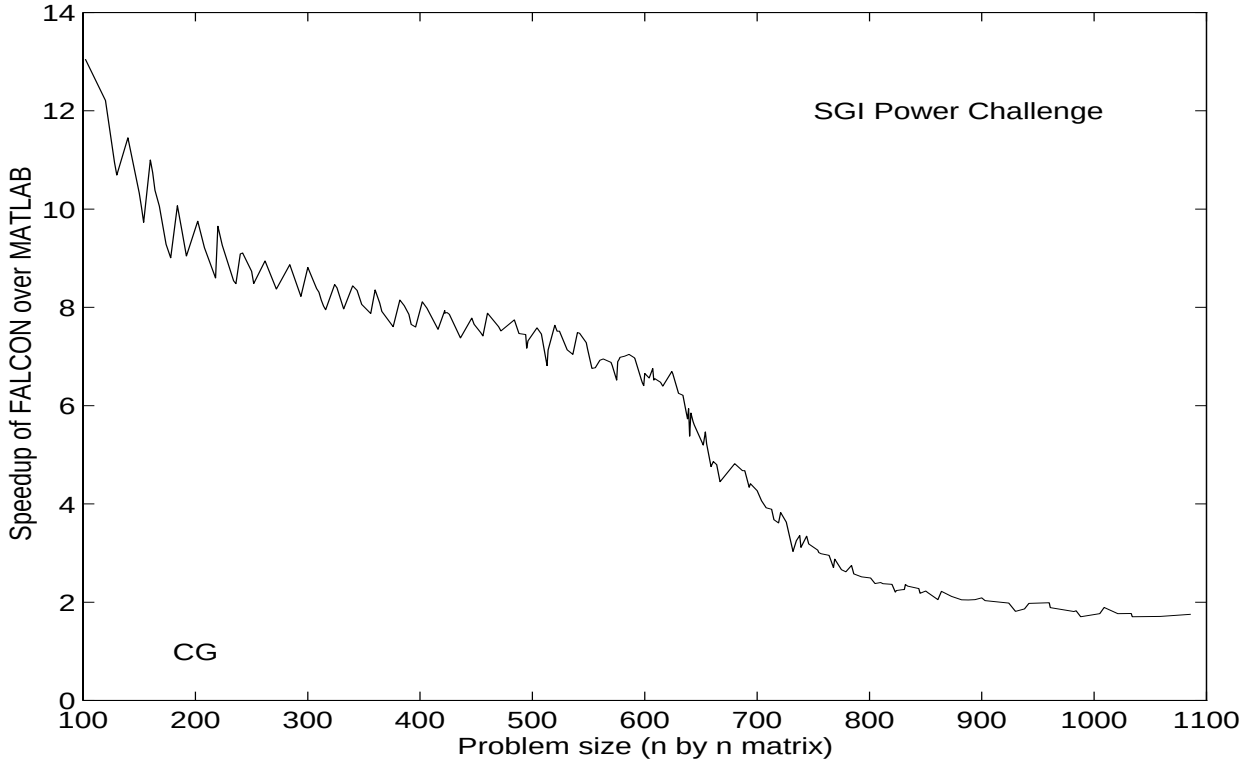

Figure 7.9: CG speedups on the SGI Power Challenge when varying the problem size.

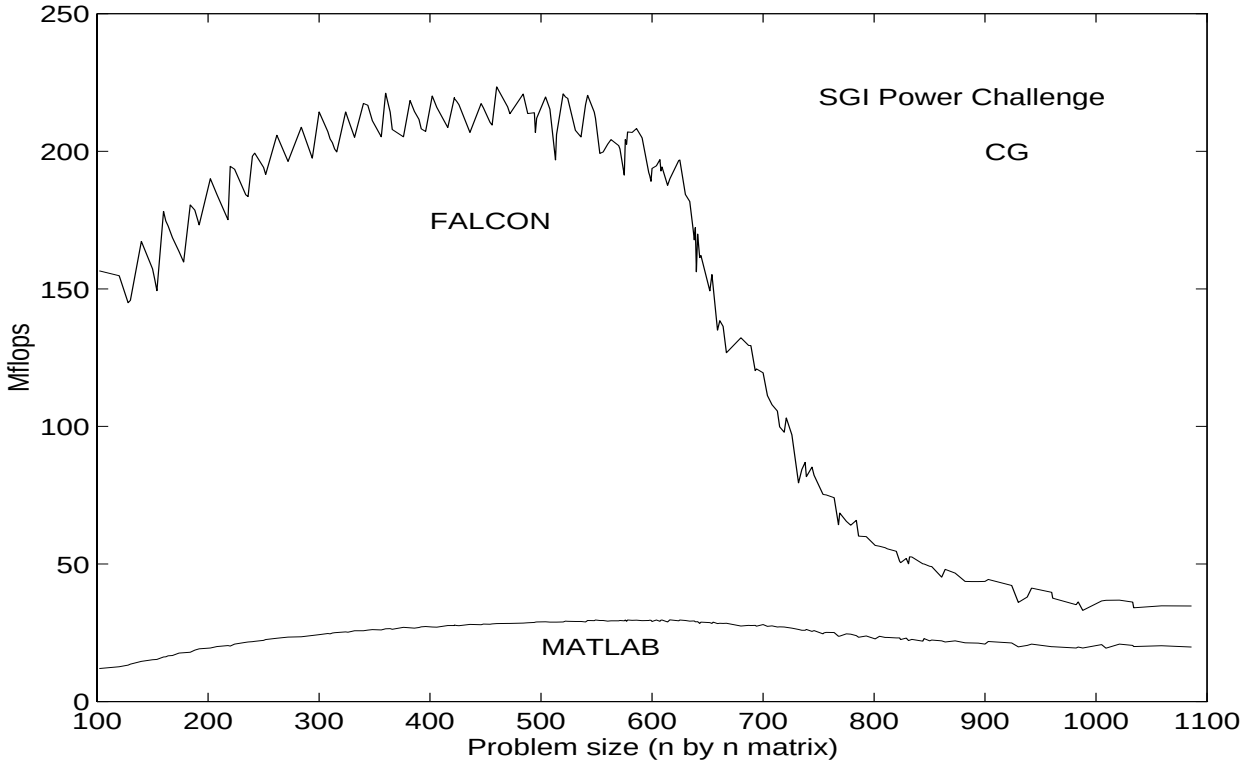

Figure 7.10: Mflops ratios for the CG program running on the SGI Power Challenge.

program. Notice that although the Mflop curve for the MATLAB interpretation is much flatter then the Mflop curve for the compiled version, the performance of the MATLAB interpretation increased by a factor of two when the matrix size changed from 100×100 to 300×300, while the performance of the compiled code increased by about one third. As the problem size increases, the matrix-vector multiplication starts to dominate the computation, and the speedup curve approaches an asymptotic value. Notice that both the compiled code and the MATLAB interpretation use the same library function to compute the $\mathcal{O}(n^2)$ matrix-vector multiplication (a BLAS-2 DGEMV). In this second region of the speedup curve, the compiled program is making full use of cache, reaching its peak Mflop ratio. The large difference in the Mflop ratio between the two curves indicates the better cache utilization by the compiled program. This better cache utilization is supported by the drop in speedup that occurs in the third region of the curve. At this region, the performance degradation due to cache misses is much more noticeable in the compiled program than in the interpreted code. Finally, in the fourth region of the speedup curve, the problem size is so large that the program is running practically out of memory. At this point, the speedup curve reaches another asymptotic value.

One possible reason for the better cache utilization from the compiled program is the interpreter's memory requirements. The compiled code uses the instruction cache for the program, while the interpreter uses the data cache for both the data and the internal representation of the M-files being used. Hence, it is likely that with the increase in problem size, the overhead of the interpretation will increase since the instructions of the M-file will have to be accessed from memory. Unfortunately, due to the complexity of the SGI Power Challenge, and the lack of single-user time and of access to the internals of the MATLAB interpreter, it is very difficult to prove this assumption.

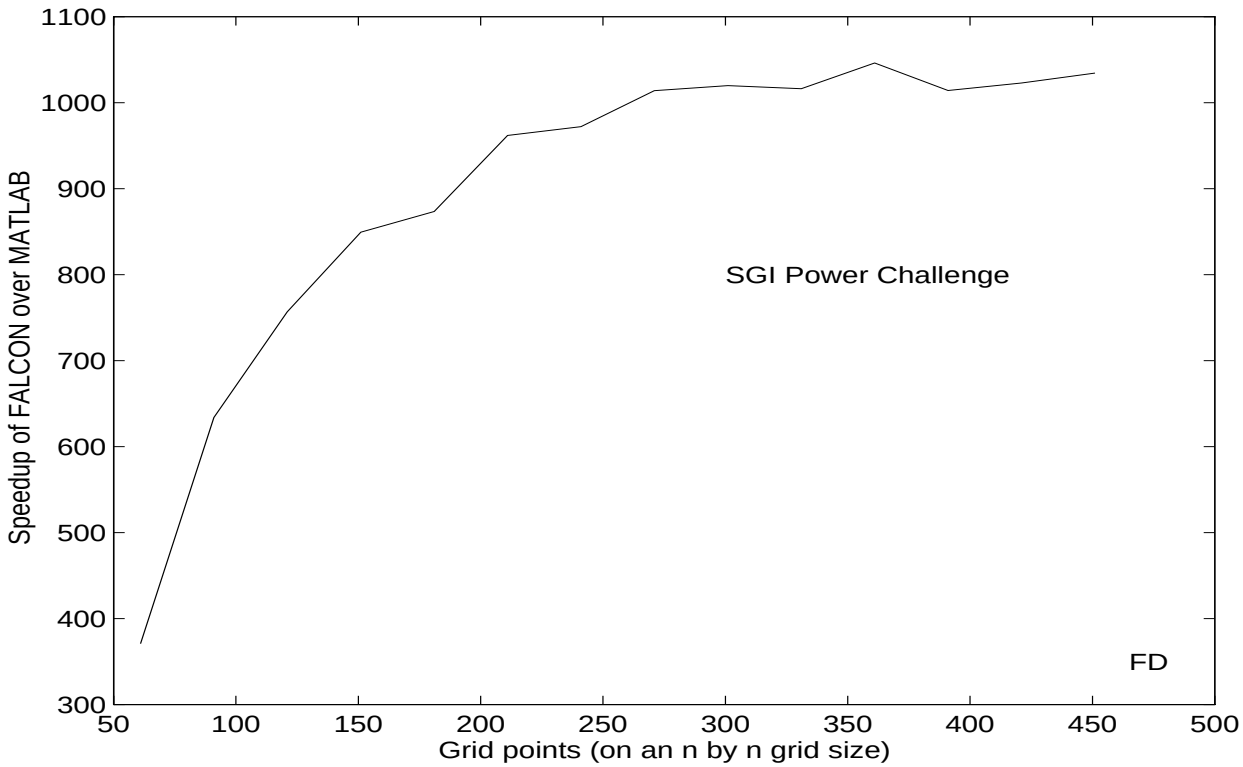

Figure 7.11: FD speedup on the SGI Power Challenge varying the grid size.

7.4.2 Analysis of Elementary-operation Intensive Programs

For the analysis of elementary-operation intensive programs, we run the program FD on an SGI Power Challenge varying the grid size from 31×31 grid points to 451×451 grid points. The speedup curve of the Fortran 90 programs over MATLAB is presented in Figure 7.11.

As expected, the speedup curve has a behavior similar to of the speedup curve for the CG program with fixed problem size. In this case, no built-in function is used; thus, the speedup is only due to the overhead of the interpretation. After the initial loop overhead of the compiled code is amortized, the speedup reaches the asymptotic value.

7.4.3 Analysis of Memory-intensive Programs

For the analysis of memory-intensive programs, we performed three sets of runs with the AQ program 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 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 pre-allocation 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 pre-allocation of 10000 subintervals, which was sufficient for all runs. Therefore, in this case, no reallocations of the array were necessary. Figures 7.12 and 7.13 present the speedups of the compiled code over MATLAB running on the SGI Power Challenge and on the SPARCstation 10, respectively.

We observe a similar behavior on both architectures. We notice that the speedup decreases as the memory operations (data movements and reallocations) of the programs start to dominate the execution time. The speedup curves for the runs on the SPARCstation 10 appears to have reached their asymptotic value, while on the SGI they are still decreasing. The overhead of allocating and using a larger array than necessary is responsible for the smaller speedup of the runs with full pre-allocation, compared with the runs with 2500 pre-allocated sub-intervals.

In this program, although it is not possible for the inference mechanism to avoid the overhead of reallocation, by determining the correct size required by the array, the gap between the speedup curves for the runs with full pre-allocation and the runs with no pre-allocated demonstrate the importance of inference techniques for pre-allocation of arrays. For this program, this gap indicates the cost of dynamic allocation. We observe that the speedup on the SGI practically doubles when the array is pre-allocated.

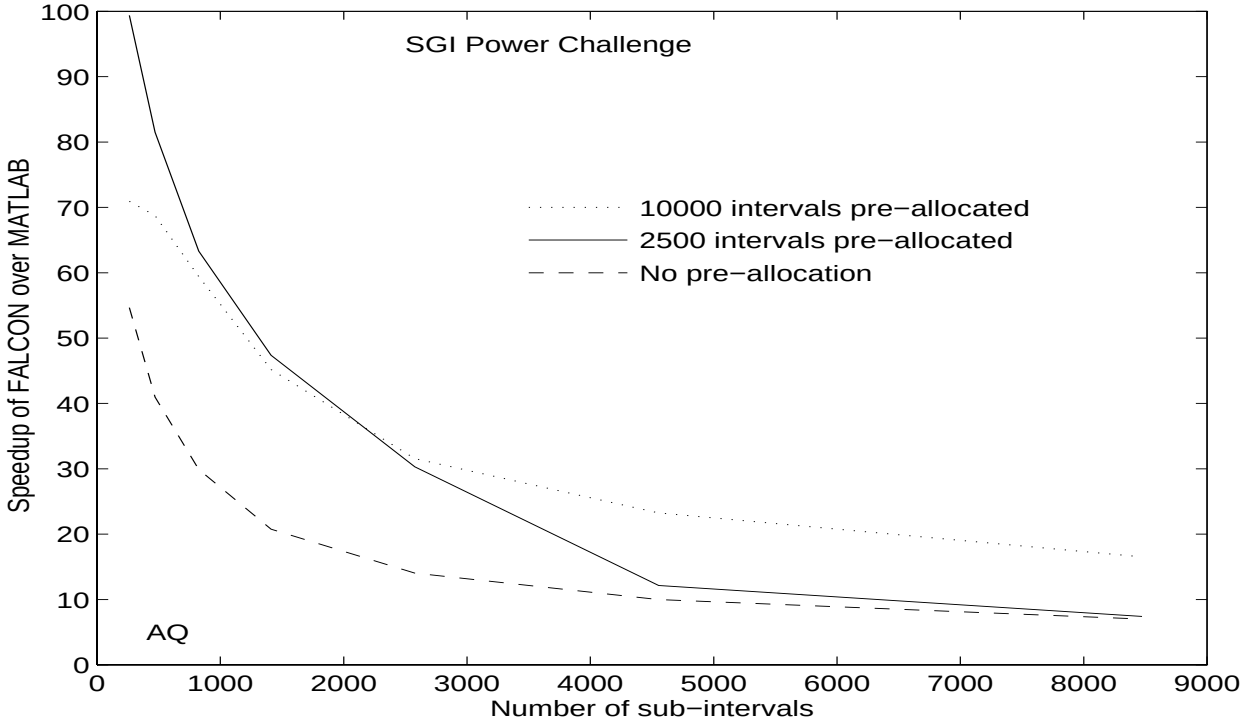

Figure 7.12: AQ speedup on the SGI Power Challenge when varying the required number of subintervals.

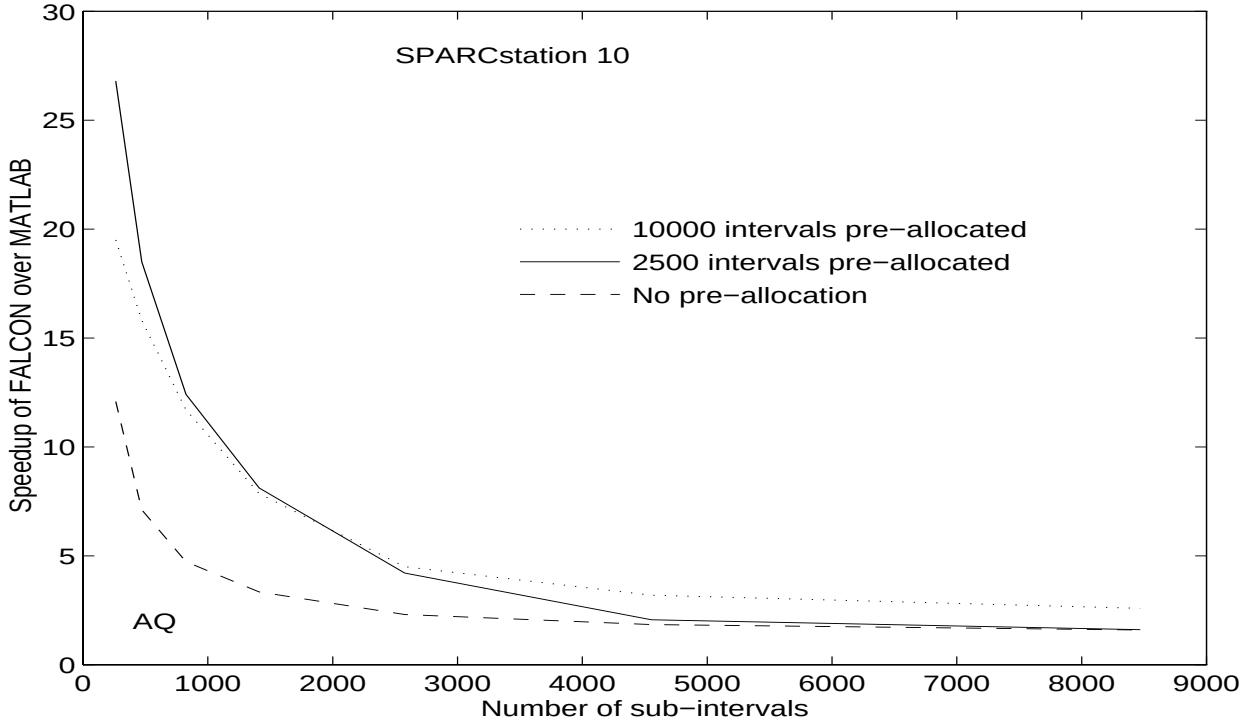

Figure 7.13: AQ speedup on the Sun SPARCstation 10 when varying the required number of subintervals.

Chapter 8

CONCLUSIONS AND FUTURE DIRECTIONS

The main goal of this thesis has been the development of inference techniques for the compilation of MATLAB programs. The MATLAB compiler presented here is part of a programming environment for the development of scientific libraries and applications, that is being developed at CSRD. By using MATLAB as the source language and producing Fortran 90 as output, this environment takes advantage of both the flexibility and power of an interactive array language and the performance of compiled languages. In order to generate code from the interactive array language, the compiler combines static and dynamic inference methods for intrinsic type, shape, rank, and structural inference. This inference mechanism is enhanced with value and symbolic dimension propagation analyses.

As shown by our experimental results, the compiled programs performed better than their respective interpreted executions, with performance improvement factors varying according to the characteristics of each program. For certain classes of programs, our compiler 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-base 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 our compiler with the performance of the programs generated by the MathWorks MATLAB to C Compiler (MCC),

we observe that in MCC generated code similar in performance to the code generated by our compiler in only one situation. In the other cases, the performance of the programs generated by our compiler was faster than the corresponding program generated by MCC, with performance difference ranging from approximately 2 to 90 times faster on a SPARCstation 10 and from 3.5 to 280 times faster on the SGI Power Challenge. This difference in performance is attributed to the more enhanced inference mechanism utilized by our compiler.

8.1 Future Work

A reasonable amount of work, both in the theoretical aspects and the implementation aspects, is still necessary to improve the performance of the target programs. This work can be divided into the improvement of sequential performance and the exploitation of parallelism.

For the improvement of sequential performance, a global analysis mechanism should be designed to enhance the static inference system. This mechanism should consider the context of blocks of statements to be able to improve the inference. Also, the symbolic dimension propagation described in this thesis should be extended to handle more complex symbolic computation. Areas needing improvement in the implementation side include the implementation of the structural inference; further exploitation of backward inference; and the implementation of cloning or inter-procedural analysis in order to reduce program size.

The integration of this compiler with a parallelizing compiler is an important step for the exploitation of parallelism. This integration includes the utilization of the high-level semantics of the array language in order to facilitate the work of the parallelizer. We envision two possible methods for this integration: an external integration with the generation of directives for parallelization; or an internal integration, by generating the internal structure of the parallelizer compiler, providing more direct information that is synthesized by the inference process.

Two other important areas of research with respect to the compilation of MATLAB are the study of techniques for the generation of code for sparse computation, and the study

of techniques for the generation of data parallel languages, such as HPF, including the automatic generation of directives for data distribution.

BIBLIOGRAPHY

[AALL93] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Amy W. Lim. An Overview of a Compiler for Scalable Parallel Machines. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 253–272. Lecture Notes in Computer Science, vol. 768, Springer-Verlag, August 1993. 6th International Workshop, Portland, Oregon.

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, 1992.

[ASU85] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1985.

[BBC+93] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1993.

[BBG+93] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. In *OONSKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 1–24, April 1993.

[BE94] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. In *Proceedings of Supercomputing '94*, pages 528–537, November 1994.

[BEF+94] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 141–154. Lecture Notes in Computer Science, vol. 892, Springer-Verlag, August 1994. 7th International Workshop, Ithaca, NY, USA.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems*, 13(4):451–490, October 1991.

[Chi86] Wai-Mee Ching. Program Analysis and Code Generation in an APL/370 Compiler. *IBM Journal of Research and Development*, 30:6:594–602, November 1986.

[Coo88] Grant O. Cook Jr. ALPAL A Tool for the Development of Large-Scale Simulation Codes. Technical report, Lawrence Livermore National Laboratory, August 1988. Technical Report UCID-21482.

[DGG+94] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. An Environment for the Rapid Prototyping and Development of Numerical Programs and Libraries for Scientific Computation. In F. Makedon, editor, *Proc. of the DAGS'94 Symposium: Parallel Computation and Problem Solving Environments*, pages 11–25, Dartmouth College, July 1994.

[DGG+95a] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: A MATLAB Interactive Restructuring Compiler. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 269–288. Lecture Notes in Computer Science, vol. 1033, Springer-Verlag, August 1995. 8th International Workshop, Columbus, Ohio.

[DGG+95b] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: An Environment for the Development of Scientific Libraries and Applications. In *Proc. of the KBUP95: First international workshop on Knowledge-Based systems for the (re)Use of Program libraries*, Sophia Antipolis, France, November 1995.

[DGK+94] D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan, and R. W. Johnson. EXTENT: A Portable Programming Environment for Designing and Implementing High-Performance Block-Recursive Algorithms. In *Proceedings of Supercomputing '94*, pages 49–58, November 1994.

[DJK93] Peter Drakenberg, Peter Jacobson, and Bo Kagstrom. A CONLAB Compiler for a Distributed Memory Multicomputer. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, Norfolk Va*, March 1993.

[DMBS79] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK User's Guide*. Society for Industrial and Applied Mathematics, 1979.

[DPar] Luiz DeRose and David Padua. A MATLAB to Fortran 90 Translator and its Effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing*, to appear.

[Gar94] Alejandro L. Garcia. *Numerical Methods for Physics*. Prentice Hall, 1994.

[GHR94] E. Gallopoulos, E. Houstis, and J. R. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science & Engineering*, 1(2):11–23, Summer 1994.

[GMBW95] K. Gallivan, B. Marsolf, A. Bik, and H. Wijshoff. The Generation of Optimized Codes using Nonzero Structure Analysis. Technical Report 1451, Center for Supercomputing Research and Development, September 1995.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[GP92] Milind Girkar and Constantine D. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), March 1992.

[GR83] A. Goldeberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, 1983.

[GR84] L. Gilman and A. Rose. *APL : An Interactive Approach*. Wiley, 1984.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[HRC+90] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, Ko Yang Wang, and S. Weerawarana. //ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines. In *Proceedings 1990 International Conference on Supercomputing*, pages 96–107, 1990.

[HWA+88] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, J. Hughes, T. Johnsson, D. Kieburtz, S. P. Jones, R. Nikhil, M. Reeve, D. Wise, and

J. Young. Report on the Functional Programming Language Haskell. Technical report, Yale University, December 1988. Technical Report DCS/RR-666.

[JKR92] Peter Jacobson, Bo Kagstrom, and Mikael Rannar. Algorithm Development for Distributed Memory Multicomputers Using CONLAB. *Scientific Programming*, 1:185–203, 1992.

[Joh] Steve Johnson. Yacc: Yet Another Compiler-Compiler. Unix Programmer's Manual - Supplementary Documents.

[Ker95] Yaron Keren. *MATCOM: A MATLAB to C++ Translator and Support Libraries.* Technion, Israel Institute of Technology, 1995.

[LS] M. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. Unix Programmer's Manual - Supplementary Documents.

[MAE+65] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *Lisp 1.5 Programmer's Manual.* The MIT Press, 2nd edition, 1965.

[Mat92a] The Math Works, Inc. *MATLAB, High-Performance Numeric Computation and Visualization Software. User's Guide*, 1992.

[Mat92b] The Math Works, Inc. *MATLAB, High-Performance Numeric Computation and Visualization Software. Reference Guide*, 1992.

[Mat92c] John H. Mathews. *Numerical Methods for Mathematics, Science and Engineering.* Prentice Hall, 2nd edition, 1992.

[Mat95] The Math Works, Inc. *MATLAB Compiler*, 1995.

[McK76] W. M. McKeeman. Compiler Construction. In *Compiler Construction, An Advanced Course.* Springer-Verlag, 1976. Lecture Notes in Computer Science, Volume 21.

[Pac93] Pacific-Sierra Research Corporation. *VAST-90 Fortran 90 Language System: User guide*, 2.1 edition, 1993.

[PGH+89] Constantine Polychronopoulos, Milind Girkar, Mohammad Reza Haghighat, Chia-Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: A New Generation Parallelizing Compiler. In *Proceedings of 1989 Int'l. Conference on Parallel Processing, St. Charles, IL*, volume II, pages 39–48, August 1989.

[Pom83] S. Pommier. *An Introduction to APL*. Cambridge University Press, New York, 1983.

[SBD+76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines - EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, second edition, 1976.

[Sch75] J. T. Schwartz. Automatic Data Structure Choice in a Language of a Very High Level. *Communications of the ACM*, 18:722–728, 1975.

[TP95] Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

VITA

Luiz Antônio De Rose was born in Porto Alegre, Brazil, on the 2nd of November, 1956. He graduated from University of Brasilia with a B.S. in Statistics in 1978. While there, he was awarded in 1976 with the Scientific Initiation Scholarship from the Brazilian National Council of Research and Development. Also, from October 1976 to May 1979, he worked for the research group at the University of Brasilia's data processing center.

From May 1979 to October 1984, he worked at the Brazilian Federal Research Center in Communications. His research was in the area of computer security and cryptology. From 1980 to 1982 he also attended the University of Brasilia, receiving his M.S. degree in Statistics and Quantitative Methods. From November 1984 to August 1987, he was a consulting analyst at the Federal Data Processing Department in Brazil (SERPRO), working with distributed systems, networks, computer security, and cryptology.

In 1987, he received a scholarship from the Brazilian National Council of Research and Development (CNPq) for his graduate studies in Computer Science. Since August 1987, he has been a graduate student at the University of Illinois at Urbana-Champaign, working towards a Ph.D degree in Computer Science. He received his M.S. in Computer Science in January 1992, working with Professor E. Gallopoulos, and received his Ph.D in Computer Science in 1996, working with Professor David Padua as his advisor. Since August 1989, he has been a Research Assistant with the Center for Supercomputing Research and Development (CSRD), except during the Fall of 1993 when he was a Teaching Assistant at the Department of Computer Science. As a TA, he received a departmental award for outstanding TA work.