Reverse Engineering Scientific Computation FORTRAN Code

Reverse Engineering of Scientific Computation FORTRAN Code

By
Olivier Étienne Dragon, B.Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Applied Science
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2006) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Reverse Engineering of Scientific Computation FORTRAN Code

AUTHOR: Olivier Étienne Dragon, B.Eng.(McMaster University)

SUPERVISOR: Dr. Jacques Carette and Dr. Alan Wassyng

NUMBER OF PAGES: ix, 72

Abstract

In this day and age, many companies struggle with the maintenance of legacy scientific software systems written in outdated programming languages. These languages use low-level control structures, algorithmic operations and cumbersome syntax that make the true meaning of an algorithm difficult to understand. To make matters worse, the process of reverse engineering the algorithm to specification often involves a considerable amount of manual work which is error-prone and time-consuming.

This thesis explores a completely automated method of reverse engineering. We apply this method to FORTRAN77 linear algebra software. This software is transformed to an extension of FORTRAN77, which we call Fortran-M. This language allows for high-level mathematical constructs such as sums, products and vector and matrix operations. To serve as a proof-of-concept for this method, we have developed a tool which uses a combination of pattern matching on the source code's abstract syntax tree to recognise low-level control structures, and symbolic analysis to determine the meaning of loops. Once a pattern has been recognised, or a loop's invariant found, we apply transformations to the syntax tree, thus creating a Fortran-M equivalent.

Acknowledgements

I would like to take a few lines of paper to thank some of the many people that have made this work possible, a humble recognition for the tremendous help they were to me. Thank you to Jacques Carette and Alan Wassyng, my supervisors, who believed in me, gave me this opportunity and helped me complete it successfully. Thanks to Spencer Smith and Wolfram Kahl for teaching me well in undergrad and for being so kind as to be examiners for my defense. Thanks to Dieter Stolle and to John Luxat for providing me with scientific computation software to test my proof-of-concept tool. A special thank you to Stephen Forrest for his incredible help with Maple, without which my prototype would not have achieved half of what it does now.

Merci à mon père Normand, pour tout, vraiment. Tes encouragements, ton oreille attentive et ton support monétaire n'est qu'un très bref aperçu de ce qui m'a permis de compléter mon travail. J'espère bien pouvoir te rendre la pareille un jour. Tu est un père extraordinaire! Merci à tous mes amis et à ma famille qui, malgré la grande distance géographique, sont restés près de mon coeur. Votre amour et amitié me sont infiniment précieux.

At last I come to you, my dear wife, Erika. Despite the long hours spent working, despite the extra two years it took me to further my studies, you endured, persevered, and somehow found the strength to support me in unimaginable ways, still hanging on to the dream of soon heading back home with your husband and start a "real life." You have been my pillar of strength for the past six years; my sunshine during the day, my moonlight at dusk. Thank you, Erika, for your love and support. I truly look forward to spending the rest of my life with you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we explore the field of reverse engineering computer programs. By reverse engineering we mean the abstraction from code into higher level constructs, such as creating a mathematical sum from a loop. The goals of reverse engineering are to improve the understandability, readability and maintainability of software. Our goals are no different. We aim to achieve these goals on a restricted set of software; specifically, legacy scientific computation code written in FORTRAN 77.

Our principal motivation in delving into this endeavour is the maintenance of the plethora of scientific computation software that was written 20-30 years ago that is still used today in many production systems. These software programs work well and have done so for many years. Nevertheless they are often extremely difficult to maintain. The reasons why such code is difficult to maintain are many: lack of documentation, inaccurate or missing design, original programmers having moved on; these are only some of the obvious ones. Thus our end goal is to improve the documentation of the software by abstracting away algorithmic details while retaining the code's original purpose. This would make the code easier to read and understand for the people who have to maintain these systems. Eventually, such documentation could also ease migration of these systems to a more modern language if the need arose.

Reverse engineering of scientific computation software offers numerous advantages. By choosing Fortran as input to our method, in particular FORTRAN 77, we give ourselves access to an incredible amount of code in domains such as numerical analysis, physics and engineering. Having vast amounts of test data further assists in validating our results and improving our method. Another important advantage is that scientific computation code tends to contains more linear algebra algorithms and less behavioural code. Implementations for the latter vary at great lengths and as such are challenging to represent by abstract mathematics. Using scientific computation code also restricts the scope of our end results, as we can assume these will

mostly involve mathematical formulae involving scalars, vectors and matrices. But most importantly, the large amount of legacy scientific code requiring maintenance today creates a need for this work, and a need for a tool that can implement this method. This further motivates the usefulness of this project.

We aim to achieve translation of legacy Fortran programs to an extension of the language which we call Fortran-M. This extension of Fortran adds high-level mathematical constructs usually not witnessed in general purpose programming languages, such as sums and products, as well as operations on vectors and matrices; constructs more familiar to a mathematician than to a programming language designer. Ideally we want to achieve a complete translation of all linear algebra algorithms into formulae. In practice we have have achieved a significant degree of success, however far from complete. In the future, we would also like to be able to bring the code to a mathematical level high enough that particular scientific equations, specific to a particular domain of science such as physics, can be easily recognised simply by looking at the reverse engineered result.

Another major goal of this thesis is to develop an analysing program that implements the discussed reverse engineering method, in order to validate our ideas and research. This tool is completely automated. There is no user input needed at any stage of the method beyond what is required to launch the analysing program and to specify the input code to be reverse engineered. In addition, our method and tool make no use of external data or knowledge such as documentation, specifications or comments. It only uses the information provided by the Fortran code itself. Although the possibly very useful information contained in comments and external documentation is forgone, we can achieve a greater automation by only looking at the code. Hence we have created a prototype analysing program which we call F-Fort. F-Fort produces two different outputs: Fortran code with added type information, equivalent algorithms and additional comments, and the reverse engineered output in a reader-friendly documentation format.

1.1 Field Survey and Related Work

Reverse engineering is a very broad term that covers a large area of work and research. In fact, Chikofsky and Cross have identified six fairly distinct fields which have created confusion in discussions about what reverse engineering really is [8]. The six fields are

forward engineering, also known as engineering, is the process of producing abstract requirements and design, down to an implementation.

reverse engineering is the opposite of forward engineering, that is to produce a

more abstract, high-level view from an implementation.

redocumentation aims to provide alternative view of a system at a similar level of abstraction to improve understanding (e.g. control flow and data flow graphs).

design recovery uses external knowledge and deduction to recreate a system's architecture, such as redefining modules and their interactions.

restructuring, also known as refactoring, aims at transforming a system at the same abstraction level while preserving external behaviour.

reengineering often involves some reverse engineering in order to reassess the design, and is followed by forward engineering to recreate a functional system.

Without going into further details about what each of these fields covers, we can assert this project's purpose better by putting it in context of these fields. Based on criteria defined by Chikofsky and Cross, this project would fall into the reverse engineering category, with some of its goals reaching redocumentation, and with the possibility of being the basis for a useful automated reengineering tool.

We say this project is mostly reverse engineering because of the type of abstraction, and what is abstracted. The abstractions we perform, converting programming language constructs to more mathematical ones, is what differentiates reverse engineering from redocumentation in [22, 31]. While most redocumentation tools focus on giving alternate views of the code itself but without making any modifications, we choose to transform the code to a higher level of abstraction; in other words, removing some operational details while maintaining the same denotation. However we do employ redocumentation techniques for some aspects such as making the type of every identifier explicit. Moreover our project shares a common goal with redocumentation: improve the understandability of code for maintainers.

What we abstract additionally makes it clear that we are not attempting design recovery [5, 6, 20]. We instead focus on a finer granularity than subroutines, modules and architectural design. Our attention is on statements inside subroutines, and our method takes no information from the organisation of modules.

The reason we mentioned that this method may be used in reengineering is that it may be possible to do some forward engineering program transformations from Fortran-M back into FORTRAN 77. In fact our proof-of-concept tool attempts to do this and succeeds in a number of cases. As this is not the main focus of our method and merely an interesting by-product, we shall not spend more time discussing it.

The goals and ideas of this thesis are similar to those of Ward with the WSL and FermaT system [37, 38]. FermaT is a formal transformation system based on the intermediate language WSL, which stands for Wide Spectrum Language. WSL

contains a large variety of programming constructs, ranging from low-level assembly-like instructions to high-level specification expressions. We borrow from these ideas and apply them specifically to linear algebra and scientific computation software written in Fortran.

Another factor that distinguishes our work from that of Ward, and what we think is one of our most important assets, is that our method is being created with complete software automation in mind. That is, everything in our method is to be performed automatically by a software tool without any user help as far as the reverse engineering method itself is concerned. Tools like Ward's maintainer's assistant [35] and FermaT use heuristics to automate a considerable portion of the method. They also automatically perform the transformations specified by the user, thus reducing the errors that can be introduced by incorrect transformations. On the other hand they still leave some work for the users to perform. In fact Ward argues that an automated process is not sufficiently powerful to be used as sole method of reverse engineering [36]. For this project we decided to push the limits of automation and see how far one could go if user input was unavailable or impractical.

A number of attempts have been made in automating the reverse engineering task of finding and performing transformations. These generally look for more intelligent ways to select the order in which transformations are applied which theoretically should lead to a better result [36]. Taking the path less travelled we chose a symbolic analysis of loops [16] approach to automatically recognise higher level computations, thus further automating the process. Other attempts have been recently made to use symbolic analysis (or interpretation) in the hopes of improving the results of automated reverse engineering [9, 34, 40]. In fact, their research has motivated many of our decisions. Also noteworthy are other techniques that have been used for program comprehension of Fortran code, namely partial evaluation [7].

1.2 Structure Of The Thesis

In Chapter 2 we begin by motivating our method and design decisions. In Chapter 3 we explain how we extract information from the Fortran syntax, and eliminate assumptions made by language compilers. Information such as variable types, subprogram inputs and outputs, parallel statements and initial variable value is determined. We delve even deeper into the reverse engineering process in Chapter 4 as we attempt to find closed-forms for loops, and from those expose patterns of linear algebra computations. Finally, in Chapter 5 we take a closer look at our proof-of-concept tool, with which we tested our method.

Chapter 2

Reverse Engineering Method Overview and Design

We begin our reverse engineering method by parsing the Fortran code to obtain an abstract syntax tree (AST). This tree still contains several Fortran specific programming constructs. It has however been stripped of superficial syntax elements that can be expressed by the tree structure. A typical example of this is for arithmetic operator precedence in arithmetic expressions. The expression $1 \times 2 + 3$ is generally accepted in mathematics as being equivalent to $(1 \times 2) + 3$ where the multiplication takes precedence over the addition. The addition is thus performed on the result of the multiplication and its second argument: 3. The parsing process accepts either expression creating the same tree structure. In the former it disambiguates the operator precedence, and in the latter case removes the parentheses. The tree structure is shown in Figure 2.1. This parsing process can be seen as a transformation of a uniform stream of characters from the source code file to an abstract syntax tree. Similarly to all other transformations performed in this method, it preserves the original semantics.

We use a syntax tree for the remainder of our method as it is much easier to

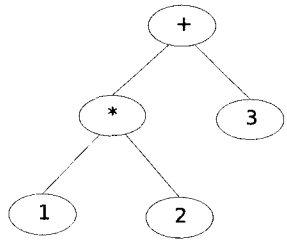


Figure 2.1: Abstract syntax tree of an arithmetic expression

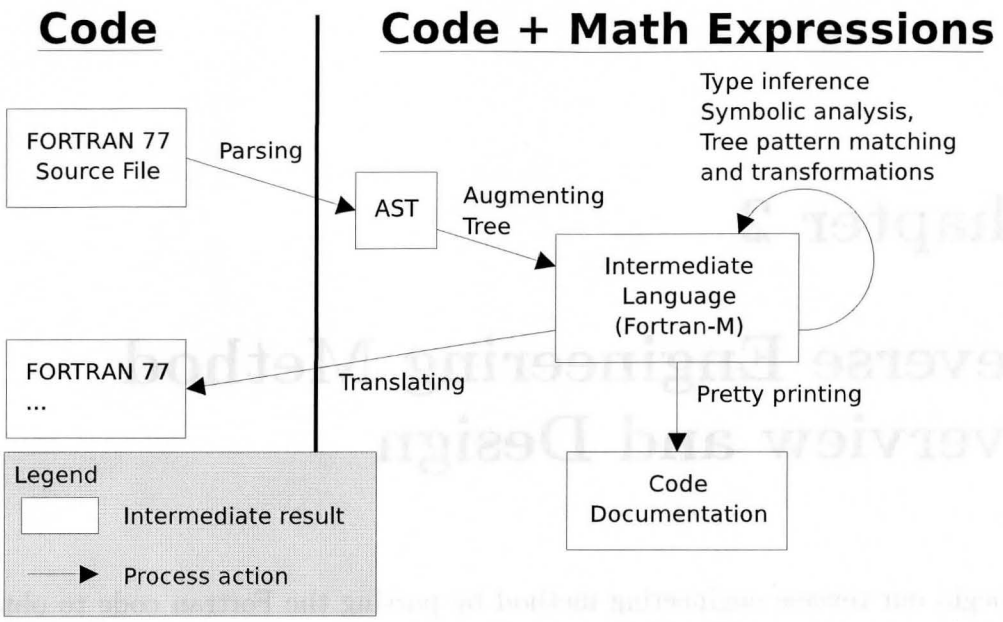


Figure 2.2: Our reverse engineering process

handle programmatically. Thus we begin the second stage of our process, which is to iteratively traverse the tree and apply rewriting rules to it: match patterns in the language and symbolically analyse loops, and accordingly transform the input code.

Figure 2.2 shows the different intermediate results of our method and the transitions between them.

2.1 Tree Pattern Matching

The notation found in Section 2.2.1 and Appendix A describes a tree language grammar. Throughout our process we use the concept of a tree grammar to perform pattern matching. However it is not used in a conventional pattern description way, like for a regular expression. It is rather akin to a programming language parser that uses a stream of tree nodes as tokens [28]. In fact we replace the conventional combination of tree-walking and pattern matching with tree parsing, which has several advantages for us: it gives us better context information and more importantly, it is thorough [29]. A pattern matching method will simply ignore patterns it does not recognise. A tree parser however must match everything in between. We can then perform transformations on select matches. We find that this reduces the number of overlapping patterns with the added advantage of using directly Fortran-M's language

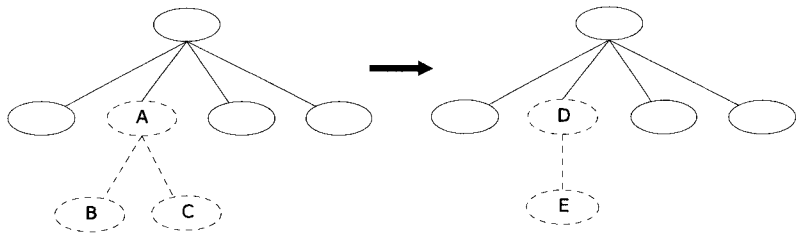


Figure 2.3: Rewrite rule

definition.

As an example, Fortran statements must appear in a certain order [3, 11, 12]. Using simple regular expressions, matching would be tried over the whole tree, as opposed to only where a certain statement containing the pattern is allowed.

The pattern matched is a subtree within the language, that is the Fortran-M syntax tree. Figure 2.3 illustrates the rewriting actions: parsing the tree identifies the #(A B C) subtree as a pattern we wish to transform. In this case, the desired action is to replace the subtree with another: #(D E).

2.1.1 About Transformations

The goal of these transformations is to take low-level Fortran code filled with algorithmic details and transform them into mathematical expressions at a higher level of abstraction. At first only low-level Fortran-specific patterns match the branches in the tree. As such, the process begins with simple transformations and traverses the syntax tree multiple times in order to expose more complex patterns. For this to work, it is important that the rewrite rules compose nicely with one another, which is a non-trivial task. By keeping the patterns as small as possible we believe that we may be able to achieve, at least in practise, confluence. Assuming patterns compose nicely allows us to reduce the size of patterns.

Take for example this simple matrix multiplication algorithm in pseudo-code [19]:

$$
\begin{aligned}
&\textbf{for } i = 1 : m \\
&\quad \textbf{for } j = 1 : n \\
&\quad\quad \textbf{for } k = 1 : p \\
&\quad\quad\quad C(i,j) = A(i,k)\, B(k,j) + C(i,j)
\end{aligned}
$$

We could try to match the larger pattern with all three for-loops. By breaking down the patterns we can recognise that the innermost loop is actually a dot product, and that we can transform this code into the following:

$$\begin{aligned}&\textbf{for } i = 1 : m\\ &\quad \textbf{for } j = 1 : n\\ &\qquad C(i,j) = \vec{A}(i,) \cdot \vec{B}(,j) + C(i,j)\end{aligned}$$

From this we establish that the j-loop is a vector update operation. The dot product becomes a vector-matrix multiplication and we obtain:

$$\begin{aligned}&\textbf{for } i = 1 : m\\ &\qquad \vec{C}(i,) = \vec{A}(i,)\, B + \vec{C}(i,)\end{aligned}$$

We finish by recognising that the i-loop is similar to the j-loop. This leads to the final and desired form:

$$C = A\,B + C$$

This is an example of patterns that compose nicely. That is, the result of composing the patterns and transformations is the same as if we had tried to match the larger pattern. In some cases it may even be more successful: where the larger pattern may fail, a fraction of the smaller ones could succeed. We explore this example in more detail using our method in Section 4.5.5.

The most important thing to note about these transformations is that during the course of our process, no information is lost. Information is only added. In that respect, code comments remain as part of the syntax tree. Not doing so can result negative consequences as code comments often contain a wealth of information. This information can be particularly useful to maintainers. In the case where no patterns are recognised, no transformations are performed. This further ensures safekeeping of any information that may be contained in the code.

2.1.2 Notes On Precision

Our method makes a significant simplification with regards to precision. Since computers have a finite representation of real numbers, often stored as single or double precision floating point, arithmetic operations will carry errors. Integers may also carry overflow errors. These errors are inherent of the input program. Nevertheless, these are missing from our output.

In order to reduce the scope of our work we perform transformations—and particularly symbolic analysis—with infinite precision. This deviates from the *actual* behaviour of program, for the reasons outlined above. In the future we would like to take precision into account and represent it in some way in the output.

2.1.3 Choice of Symbolic Analysis

To empower the matching of patterns, we use symbolic analysis. In particular, this technique is applied to loop structures for which pattern matching is too fragile. To do the same work using rewrite rules, if at all possible, would require either an impractically large number of patterns, or extremely complex patterns.

```
      DO 70, J = 1, N
         DO 60, I = 1, M
            TEMP1 = ALPHA*B( I, J )
            TEMP2 = ZERO
            DO 50, K = 1, I - 1
               C( K, J ) = C( K, J ) + TEMP1    *A( K, I )
               TEMP2     = TEMP2     + B( K, J )*A( K, I )
50          CONTINUE
            IF( BETA.EQ.ZERO )THEN
               C( I, J ) = TEMP1*A( I, I ) + ALPHA*TEMP2
            ELSE
               C( I, J ) = BETA *C( I, J ) +
     *                     TEMP1*A( I, I ) + ALPHA*TEMP2
            END IF
60       CONTINUE
70    CONTINUE
```

Figure 2.4: `ssymm` multiplication algorithm

Take for example the matrix multiplication algorithm from the BLAS [14] `ssymm` routine for multiplying an m-by-m symmetric matrix A with an m-by-n matrix B ($C \leftarrow \alpha AB + \beta C$), shown in Figure 2.4. Besides being obfuscated by temporary variables, the algorithm also contains an if-then-else optimisation to avoid an extra costly multiplication and addition. These factors make recognising this piece of code as a matrix multiplication algorithm rather difficult; unless we have a very complex pattern to match this exact form. This example demonstrates the fragility of using only pattern matching to recognise higher levels of abstraction. Although our symbolic analysis implementation is not powerful enough to process this complex case, as explained in Chapter 4, we believe symbolic analysis could yield the desired results.

Symbolically analysing loops results in the abstraction of algorithms to the level of mathematical recurrences. This improves the generality of patterns we can use to recognise a specific algorithm, thus significantly reducing the number of patterns

needed. We can also try to obtain the closed-form of the recurrences, further abstracting away from the algorithm. These recurrences and closed-forms become new patterns that can then be caught by our rewrite rules.

2.2 An Intermediate Language

Most compilers these days use the concept of an intermediate language [1]. An intermediate language is one which is usually different from the input to the compiler but also differs from the assembly language used before compiling to binary. The compiler can then be separated into two major parts: the front-end and the back-end. The front-end translates an input language into the intermediate one, and then the back-end translates the intermediate language into the target assembly language. This allows compilers to take advantage of the same back-end for different front-ends, and thus different input languages. Or similarly, to generate different hardware platform output for a single input language. It also possesses the advantage of reusing a so called middle-end for aspects like generic optimisations, applied in all cases to the intermediate language.

As part of the development of our method, we have been inspired by such an approach. However due to the time and labour intensive nature of this project, we decided it would be best to restrict the scope of the project and create an intermediate language that kept strong ties with its single input language, FORTRAN 77. Since choosing FORTRAN 77 provided us with more than enough software to work with, we did not see this as a drawback.

Having an intermediate language, as compilers do, was not the most important reason for the creation of Fortran-M. What we mostly needed was a language that would support constructs at a higher level of abstraction than Fortran. We needed ways to express mathematics and linear algebra operators. Extending Fortran with such constructs gave us the abstraction, freedom and power of expression we were looking for.

We made another important decision concerning this intermediate language: since our desired output should be abstract, we wanted this language to remain at a high-level. This avoids having to translate to a low-level bytecode-like language and back to high-level formulae again. It also lets us perform transformations in a high-level language.

Finally, we decided that our intermediate language would be a tree language. This greatly simplifies our task as we already obtain an abstract syntax tree from the parsing process. It was further found that using a tree language eases the task of performing transformations on the language, as trees lend themselves well to mathematical reasoning and programming.

2.2.1 Notation

Here we briefly describe the notation used for this tree language. We use Extended Backus-Naur Form (EBNF [23]) in Appendix A to describe the syntax of our notation. The syntax of this notation was borrowed from ANTLR [30], and resembles some of the popular variants of EBNF. Here is an example of the language's notation,

tree : #(parent label:child1 (child2)+ #(child3 grandchild1)) | alt ;

The #(...) notation identifies a set of acceptable subtrees with the root being the first node, and its children following. Thus the *parent* node has for children *child1*, the possibly multiple *child2*, and *child3*. Concatenation, denoted in EBNF by ',', is implicitly defined by adjacent symbols. We define concatenation in our notation's context as two nodes being close siblings, that is they have the same parent and are next to one another in the node ordering. Figure 2.5 shows the pictorial equivalence of the *tree* rule which also illustrates the concatenation concept.

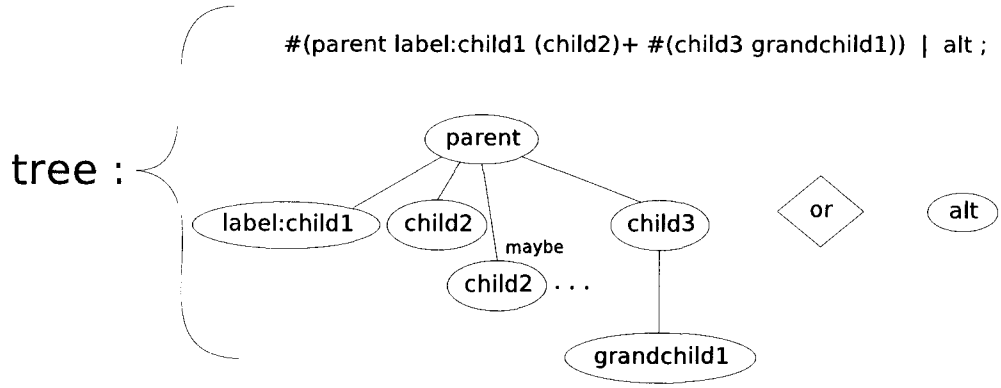


Figure 2.5: Equivalence example between the *tree* rule and a set of syntax trees

Nodes can be given labels such as "label" for *child1*. The subtree labelled with this name can later be referenced by its label. As a convention, terminal symbols are all in uppercase while non-terminal grammar rules are all in lower case. The '?' operator indicates optionality (zero or one). The '*' operator indicates optional repetition (zero or more times). The '+' operator indicates mandatory repetition (one or more times). A '.' signifies a wild-card terminal symbol that will match any subtree. Finally a '|' portrays alternatives.

2.3 Assumptions About The Input Source Code

In an attempt to restrict the scope of the project, as well as focus specifically on reverse engineering, we decided to make some assumptions about the input to our method. This allows us to ignore code that uses non-standard compiler specific extensions. In our experience, this has not reduced the amount of code available to us that our method works with.

Our hope is that the software to be analysed would be valid ANSI FORTRAN 77 code. Verifying the exact compliance of the source code is in itself rather difficult as there are no known compilers that implement the standard strictly, without any extensions or slight modifications. Nevertheless we settled on the GNU Fortran compiler `g77` to be our benchmark. This compiler is said to work for the GNU Fortran language, which is said to support a superset of ANSI FORTRAN 77 [18].

Our assumption is that that any source file that prompts the compiler to abort compilation due to errors is to be ignored. Cases where the compiler gives warnings about the source file should be looked at on a case-by-case basis as some warnings are irrelevant to our reverse engineering work. Figure 2.6 is an example of code which produces two warnings, shown in Figure 2.7, which we can ignore. The code does compile and run without ill effects. The warning about `d(3)` seems alarming. However we note that the Fortran does not verify or impose restrictions on the specified size of one-dimensional arrays. A convention has been to use "1" as a dummy size for all parameter arrays . Another convention is to use "*" to avoid confusing 1 for the actual size of the array (or perhaps simply to quiet warnings from the compiler).

Last but not least, we also assume that the input code performs its intended behaviour. That is, our method takes the input "as is." If the input code contains errors, for example casting reals to integers, or off-by-one errors, the result from our process should reflect and still contain those errors; perhaps even make them more obvious.

```
      program main
      integer a(10), b
      double precision c
      common /block/ b, c

      call array(a)
      end

      subroutine array(d)
      integer d(1), b
      double precision c
      common /block/ b, c

      d(3) = 5
      end
```

Figure 2.6: Fortran code for which g77 gives two warnings

```
warnings.f: In program 'main':
warnings.f:4: warning:
           common /block/ b, c
                  ^
Initial padding for common block 'block' is 4 bytes at (^) -- consider
reordering members, largest-type-size first
warnings.f: In subroutine 'array':
warnings.f:14: warning:
           d(3) = 5
             ^
Array element value at (^) out of defined range
```

Figure 2.7: Warnings emitted by g77 on code in Figure 2.6

Chapter 3

Harnessing Fortran

FORTRAN 77 contains numerous archaic programming constructs and hidden rules. One of our principal goals is to replace those archaic constructs and explicitly acknowledge assumptions made by compilers on aspects such as variable types. In fact most of the work described in this chapter is also done by Fortran compilers. The other part of the work consists of very small transformations which aim to simplify the syntax tree as well as provide some abstraction. Such transformations include merging subprogram types, removing local variable initialisation statements and identifying sequential assignments that can be parallellised.

3.1 The Gathering Of Information

Our method to gather the necessary information includes several steps. We begin by merging the different subprogram types (program, subroutine and function) into a single, aptly named subprogram. We then determine the explicit type of all symbols, whether they are subprogram inputs, outputs or updates, local variables, external subprograms or intrinsic functions. We also identify constants, as well as the Fortran artifacts known as statement functions. By subprogram updates we mean a subprogram parameter that is both used as input and output. By intrinsic functions we mean the Fortran built-in functions.

Additionally, the process outlined in Sections 3.1.2 to 3.1.5 is sequential. Each step assumes the information found during the previous steps. This dependency is necessary as, for example, finding whether an identifier is an input or an output to a subprogram relies on the knowledge of which identifiers are argument parameters. We also need to have the subprogram transformed so that it can store information about symbols.

3.1.1 Unsupported Fortran Statements

Before we begin, allow us to be more specific about which Fortran statements we do not handle. We do not attempt any pattern matching on them, and do little other than parse them into the syntax tree. The foremost reason for ignoring these statements is the difficulty to reverse engineering their real meaning, and the repercussion on our result being too unpredictable.

Future consideration will likely impose treatment of these statements. This should not invalidate our approach as information is never lost, and always added. By determining some high-level purpose for these statements, we simply further our understanding of the code, without contradicting previous findings. The following are the statements we ignore:

- equivalence
- blockdata
- entry
- save
- pause
- statements related to I/O (format, print, read, write, open, close, backspace, inquire, rewind)

3.1.2 Merging The Subprogram Types

In Fortran there are three different types of subprograms:

program is what is often referred to as the "main" program, that is it determines the beginning of execution when an application is run. Programs can be optionally named, but don't have parameters like other subprogram types. There can only be one such subprogram in a compiled binary.

subroutine is a named subprogram that must be accessed via a "call" statement. Subroutines can have parameters used for input, output, or both.

function is a named subprogram much like a subroutine with the exceptions that it must have a return value. It can be used as part of an expression (return value), and it cannot be accessed via a "call" statement.

The distinction between subroutines and functions can be misleading. This is due to the fact that functions, just like subroutines, may have side effects. This in turn is due to Fortran using pass-by-reference for argument parameters. Thus we use a metaphore analogous to the C programming language and treat all subprogram types the same. We think of programs as functions with inputs from the operating system and a return value to the operating system, and think of subroutines as functions without a return value that can only be used in a call statement, and thus cannot be part of an expression. Another advantage of this merge is the simplification of the syntax tree. It allows us to represent all subprogram with the same tree structure. The different trees for the subprograms are described by the following grammar:

program : #("program" (n:NAME)? SUBPROGRAMBODY) ;
subroutine : #("subroutine" n:NAME (NAME)* SUBPROGRAMBODY) ;
function : #("function" (*type*)? n:NAME (NAME)* SUBPROGRAMBODY) ;
type : "real" | "precision" | "integer" | "complex" | "double" | "logical" |
#("character" (#(STAR ICON))?) ;

Of particular note here is that "double" is used for the Fortran "double complex" type and that "precision" represents the Fortran "double precision" type. The NAME nodes labelled "n" above are the given name of the subprogram. In the case of subroutines and functions, the subsequent NAME's are the subprogram argument parameters. These are all transformed into trees of the following grammar:

subprogram : #(SUBPROGRAM (*statements*)+) ;

There are a few things that can be noted from the tree grammar above. Where we had three different syntax tree forms before, we now have one. The three subprogram types have been merged into one for the reasons outlined above. Subprograms' names, subroutines' and functions' parameters, and functions' output have been removed from the syntax tree. The main reason for this is that we want to keep in the tree only nodes that pertain to executable code. Therefore any subtree pertaining to things like subprogram parameters, typing of variables, dimensioning of arrays, etc. are removed from the syntax tree. This information however is not discarded. We keep track of the subprogram's name, it's original Fortran type, each argument parameter's name and their calling order, and, in the case of functions, the type of the return value. This information is recorded as satellite data associated with each subprogram. Lastly, the statements which were children of SUBPROGRAMBODY are now children of the newly created SUBPROGRAM node. The example below shows the syntax tree before and after the transformation for the code in Figure 3.1.

before : #("function" "integer" f:NAME (a:NAME b:NAME)
#(SUBPROGRAMBODY #(ASSIGN #(STAR a:NAME b:NAME)))) ;
after : #(SUBPROGRAM #(ASSIGN #(STAR a:NAME b:NAME))) ;

```
integer function f(a,b)
f = a * b
end
```

Figure 3.1: Simple subprogram

3.1.3 Analysing Specification Statements

In this step of our method we first use explicit Fortran knowledge embedded within the syntax tree to determine the types of identifiers. Any identifier left without a type is given one based on Fortran's IMPLICIT rules; however in some cases we have to resort to some complex logical deductions, as it is not simple to differentiate between a variable (array), an external subprogram, a statement function, an intrinsic function or a subprogram passed as a parameter to a subprogram.

So we begin by looking at the explicit specification statements in the syntax tree. These specifications are recognised using tree pattern matching. The tree pattern matching works in such a way that a terminal symbol within a pattern may not be a syntax tree leaf. And so a terminal symbol in a pattern will match a subtree of which parent node contains that symbol. Here are the statement specification patterns,

implicit : #("implicit" (#(*type* ('(' (*implicitRange*)+ ')'))+)+) ;
dimension : #("dimension" (NAME)+) ;
typeSpecification : #(*type* (NAME)+) ;
parameter : #("parameter" (#(EQUALS NAME expr))+) ;
intrinsic : #("intrinsic" (NAME)+) ;
external : #("external" (NAME)+) ;
common : #("common" ((DIV (block:NAME DIV | DIV) (NAME)+)+ |
(NAME)+)) ;
data : #("data" (*dataEntity*)+) ;

where the following subrules apply:

implicitRange : *letter* | #(MINUS *letter* *letter*) ;

dataEntity : *dataItems dataValues* ;
dataItems : #(DIV (*dataItem*)+) ;
dataValues : #(DIV (*dataValue*)+) ;
dataItem : *varRef* ;
dataValue : *expr* ;

These statements are dealt with on an individual basis. We gather information for each statement and keep track of it as satellite data for each subprogram.

implicit: The `implicit` statement in Fortran specifies the type of identifiers which do not appear in an explicit type specification statement. These identifiers are typed implicitly according to their first letter. Without an implicit statement that overrules the default behaviour, identifiers starting with 'A' to 'H' or 'O' to 'Z' are of type real while those starting with 'I' to 'N' are of type integer. We use this rule to determine the type of symbols which do not appear in an explicit type specification statement. The implicit types and respective letter ranges are recorded as part of the subprogram.

dimension: The `dimension` statement is used to specify the number of dimensions and their sizes for arrays. It does not specify the type of the array elements. We record the name of the array, its dimensions and sizes. The type can be left implicit or specified before or after the dimension statement. A symbol inside a dimension statement can either be a parameter, previously found inspecting the subprogram statement, a global variable if it appears in a common block, or otherwise a local variable. The scope is not determined until all the specification statements have been analysed.

type specification: For type specification statements we simply record the name of the variable and type information. These statements may also contain array dimensioning which we also take care of just like for the dimension statement. The same classification applies for parameters, globals or locals.

parameter (constants): The Fortran `parameter` statement is used to define symbols as constants. From now on we will refer to them as constants to avoid confusion with subprogram argument parameters. For constants we record their values and types. If not of implicit type, constants must be explicitly typed prior to the parameter statement [3].

intrinsic: The `intrinsic` statement explicitly identifies symbols to be Fortran built-in (intrinsic) functions. This is very important since a built-in function identifier

not explicitly declared intrinsic may be used as a variable or an external subprogram. We record which symbol names are intrinsic and modify the tree nodes of those symbols from type NAME to INTRINSIC to indicate they are not variables.

external: The `external` statement indicates external subprograms (functions or subroutines only) which may be called by the current subprogram. We record which symbols are external and change the tree nodes with those symbols from type NAME to EXTERNAL to indicate they are not variables. We further keep track of external subprograms for all files processed by the tool. This enables us to more easily determine the input or output status of variables passed as a parameter when an external subprogram is called.

common: The `common` statement is used to share named scope across subprograms. These so called global variables are only in scope for subprograms with a common statement declaring the same common block name. Since not all subprogram share all of the variables in common statements, we must keep track of which common blocks are in scope for a given subprogram, as well as record all common blocks at the file level. If the type of a global variable was previously declared in a specification statement, the variable is simply given a global scope. It may be typed explicitly or implicitly later. It is important to note that we do not keep track of common blocks like a compiler would do. We only care for the names, their type, dimension and size; as we are neither attempting to generate working assembly code, nor trying to interpret the code on a large scale, we do not worry about block sizes and variable ordering.

data: The `data` statement is used to initialise variables and arrays with values. It may be noted that the data statement grammar previously shown is not the same as the Fortran specification. Due to the complexity of implied-DO initialisations it was decided to only transform the data statements which used the simple form shown in that grammar. Global variables may not appear in a data statement. This is the case since globals in a named common block can only appear in a blockdata subprogram, which we do not handle. Hence only local variables and parameters can be assigned a value within a data statement. The values are recorded for each variable or array appearing in the statement.

For each statement, after the information is gathered we remove the statement subtree from the syntax tree. Table 3.1 shows examples of specification statements in Fortran (left) and in syntax tree form (right) before being analysed.

We can see that the argument parameters `a` and `b` are given an implicit type of real and an explicit type of integer, respectively. As part of the common block "block1",

```
subroutine sub1(a, b)

implicit integer (e)
double precision c
common /block1/ c, d
dimension c(3), b(10)
integer b
parameter (echo = 60)

end
```

```
subroutine
|   [sub1,<NAME>]
|   [a,<NAME>]
|   [b,<NAME>]
|   [[subprogrambody],<SUBPROGRAMBLOCK>]
|   |   implicit
|   |   |   integer
|   |   |   |   [e,<NAME>]
|   |   precision
|   |   |   [c,<NAME>]
|   |   common
|   |   |   [/,<DIV>]
|   |   |   [block1,<NAME>]
|   |   |   [/,<DIV>]
|   |   |   [c,<NAME>]
|   |   |   [d,<NAME>]
|   |   dimension
|   |   |   [c,<NAME>]
|   |   |   |   [3,<ICON>]
|   |   |   [b,<NAME>]
|   |   |   |   [10,<ICON>]
|   |   integer
|   |   |   [b,<NAME>]
|   |   parameter
|   |   |   [=,<EQUALS>]
|   |   |   |   [echo,<NAME>]
|   |   |   |   [60,<ICON>]
|   |   end
```

Table 3.1: Specification statement examples

Name	Type	Usage	Scope	Dimension	Size	Value
a	real	variable	parameter	0	1	
b	integer	variable	parameter	1	10	
c	double precision	variable	global (block1)	1	3	
d	real	variable	global (block1)	0	1	
echo	integer	constant	local	0	1	60

Table 3.2: Result of the analysis of specification statements

`c` and `d` are global. The vector `c` has size 3 and is of type double precision, while `d` is implicitly typed as real. Finally, `echo` is a constant with value 60. Under the default implicit rules it should have been of type real, but because of the overriding implicit statement, all implicitly typed symbols starting with the letter "e" are of type integer. Table 3.2 shows these results.

3.1.4 Typing And Classifying Unspecified Symbols

As Fortran makes use of various implicit rules, a goal of our method is to make all these explicit. In order to do so we must look through the code's syntax tree and search for identifiers which have not yet been typed or classified. For an identifier tree node used without children we simply assume it to be a scalar variable, unless it is part of a call statement. Since there are no intrinsic functions without arguments, and granted that external subprograms and statement functions must be explicitly declared in order to be compiled properly, this is almost always a safe assumption. There is one exception, however, and that is with argument-less functions passed as parameters. For this particular case we had to modify the parser to give a special label to identifiers with an empty set of parentheses. Since Fortran uses parenthesis for both arrays and function calls, the parser cannot differentiate between them; that is unless it is used without arguments, since arrays cannot be used without indices. Therefore, for identifier symbols without children we verify if they are subprogram parameters, otherwise we classify them as locals. We then give them an implicit type based on the current implicit rule; either the default one or a different one if specified with an implicit statement.

With respect to an identifier node with children, we have four possibilities: it could be either an array, an intrinsic function, an external subprogram passed as parameter, or a statement function. In reality, however, we can rule out arrays based on the assumption that an array symbol which has not been dimensioned is not an array. In fact, `g77` gives an error in such a case so we can safely assume arrays must always be dimensioned.

With regard to defining an unknown symbol's usage, we only need to do so for

the first encounter. We can then simply change the type of the other tree nodes with the same symbol encountered further down the syntax tree, without analysing them. This is the method prescribed by Fortran, and is reinforced by the fact that symbols with the same name cannot be used in two different ways.

statement functions: Statement functions are simple one-line expression functions that can have argument parameters. Since these must be declared prior to any executable statement, we identify a statement function as an assignment with the left-hand side being a node with children coupled with the fact that arrays must be dimensioned. We record the function's name, its arguments and its expression. Since statement functions appear after specification statements the symbol may have been typed, in which case we record that type. Otherwise we use the current implicit rule to determine the type. The function's definition statement subtree is then removed from the abstract syntax tree.

parameter subprogram: In Fortran it is possible to pass another subprogram symbol as argument to a subprogram. Therefore we must identify this usage of symbols. A parameter subprogram will appear as a parameter that has not been dimensioned but that is used with arguments. In the case where the subprogram takes no arguments it would appear as something other than an identifier and we know right away it must be a parameter subprogram as arrays must be referenced with indices. When we find a parameter subprogram we simply record its name and the number of arguments it has. If the subprogram is not a subroutine (inside a call statement) we determine its return type either explicitly or implicitly.

intrinsic function: In the case of intrinsic functions, we simply do a table look-up with a list of all known intrinsic functions. If the symbol has not been identified with any other purpose, then we must assume it is intrinsic.

Figure 3.2 shows examples of implicit usage of symbols. In program `main`, the `test` function must be explicitly declared as external since it is passed as a parameter to `sub2`. On the other hand, `sub2` is not explicitly declared; implicitly it is detected as an external routine. In `sub2`, `f` is properly identified as a parameter subprogram of type real, and `real` as a local variable of implicit type real. Note that there is also an intrinsic function named `real`. In this context however it is used as a local scalar variable. This usage forbids the calls to the `real` intrinsic function within the subroutine `sub2`. The return value of function `test` is properly identified as variable `test` of type real. The statement function `stfunc` is a function taking two real arguments and returning a real value. These types are all determined using the default implicit rule stated in Section 3.1.3. Finally it is found that `mod` and `int` are used here as intrinsic functions.

```
program main
external test
call sub2(test)
end

subroutine sub2(f)
real = f()
end

function test()
stfunc(a,b) = (a * b)**2
test = mod(int(stfunc(3,4)), 5)
end
```

Figure 3.2: Example of unspecified symbols

3.1.5 Finding Input And Output Status

A very important piece of information to know about an arbitrary piece of code is the input, output or update status of the subprogram parameters [39]. An input is defined as being an argument parameter that is only read but not written to throughout the whole subprogram. An output is a parameter that is only written to (in a pass-by-reference context, such as in Fortran) and an update is one which is both read and written to.

In order to find the input, output or update status of variables, we first determine which Fortran statements constitute memory read and which constitute write. We found that assignment, READ, DO and CALL statements changed the value of variables, as well as calls to external functions within expressions. The assignment and READ are fairly obvious. In the case of the DO loop it is the loop variable defined within the statement which is updated. In FORTRAN 77, the loop variable maintains its final value after the loop has finished executing. As for external subprograms, since Fortran uses pass-by-reference we must recursively analyse the parameters of the called subprogram to determine their input or output status.

At this point all symbols must have been typed and classified so that we do not mix external functions, subprograms, etc., with variables. We then parse the tree, and for each node we keep a set of inputs and output variables. The procedure is recursive, and parents inherit their children's inputs and outputs sets. In this matter each node, and as such each statement, has a set of each inputs and outputs associated with it. This is very useful information to know later on in the process found in Section 4.2.2.

Once we have gone through the whole syntax tree we take the intersection of the two sets and classify these variables as update, if they are parameters. Following that we subtract each set from the other, thus eliminating updates, to find the pure inputs and outputs. We must be careful not to mistake a variable that is written to and then read from as an update. In static single-assignment form such a parameter is never read, thus it is not an update. As an example, parameter `b` in Figure 3.3 is an update since it is read first and then modified before the end of the subprogram. On the other hand, `a` is written to before being read and so is a pure output.

3.2 Abstracting Fortran

In this section we show some of the patterns and transformations used on various Fortran structures. These transformations follow our goal of keeping each small.

3.2.1 Parallel Assignments

In Fortran, all statements are executed one after the other in the order they appear in the source code. That is, statements are composed sequentially to obtain the final result of a computation within a subprogram. Nevertheless, it is possible that several statements do not depend on one another for data. A simple abstraction can be done by finding parallel compositions of statements.

Detection of parallel execution paths is traditionally done using dataflow analysis on a dataflow graph. While this technique is the most thorough, we can still achieve a reasonable amount of parallelism by doing far less work. We can accomplish this by looking for patterns of sequential assignment statements which do not depend on each other for data. This is simply done by observing assignment statements following one another within a code block: inside a loop or if-statement body. When an assignment's right-hand side variable is not contained in the following assignment's left-hand side expression, we know that the two are independent, and can be expressed by a parallel composition. For additional safety, when an otherwise parallel assignment statement has a code label, we ignore it because control flow jumps into the middle of a parallel block is ill defined.

We use the example in Figure 3.3 to illustrate how our method works. Neither statement 1 or 2 contains `a` or `c` in their right-hand side expression. Thus the two can be expressed as a parallel composition. On the other hand, since statement 3 depends on the values of `a` and `c` before its execution, it cannot be grouped with statements 1 and 2 in a parallel fashion. Figure 3.4 shows the result.

```
      subroutine sub(a, b)
1     a = b ** 2
2     c = b + 5
3     b = a + c
      end
```

Figure 3.3: Example of parallel assignments

```
1.   ★ subprogram sub (a, b)
2.
3.     Output parameters:
4.     real a
5.
6.     Updated parameters:
7.     real b
8.
9.     Local variables:
10.    real c
11.
12.    Executable statements:
13.    || a ← b^2
       || c ← b + 5
14.    b ← a + c
15.    end
```

Figure 3.4: Resulting transformation of parallel statements

3.2.2 Initialisation Of Variables

The Fortran language, and its compilers such as g77, do not guarantee that variables will have a default value before they are used within a subprogram. As a result, many Fortran subprograms begin their execution with assignment statements giving initial values to some of the variables. By removing such statements from the syntax tree, we simplify the tree without losing information as we record the initial value as part of the variable's type.

To keep this transformation as safe as possible, we decided that only assignments at the very beginning of a subprogram would be used for this, and only those assignments to local variables giving a literal constant value are matched and transformed. Figure 3.5 illustrates this concept. In statement 1, a is a parameter thus we do not

```
      subroutine sub(a,b)
1     a = 0
2     c = 0
3     d = b + 1
      if (b > 0) return
4     e = 1
      end
```

Figure 3.5: Variable initialisation example

process its assignment as an initialisation. On the other hand, statement 2 assigns a constant value to local variable c; hence we remove the statement from the syntax tree and record variable c's initial value. In statement 3, d is also a local variable, but the expression found on the right-hand side of the assignment is not constant. Therefore this statement is not a candidate for our transformation. Finally, the assignment to e in statement 4 is constant but because it is not at the very beginning of the subroutine we do not use it.

3.2.3 Three-Way Branch

Fortran sports a three-way branch which they call an arithmetic if statement. This statement has the following tree pattern:

arithmeticIf : #("if" (*expr*) a:LABELREF, b:LABELREF, c:LABELREF)

Such a structure is very low-level and reminiscent of assembly language constructs. First, the expression, which must be arithmetic, is evaluated and compared to 0. If the expression value is less than 0, the control flow is transferred to label *a*. If the value is 0, control jumps to label *b*, and if the value is greater than 0 the control goes to *c*.

There are a few idiomatic uses of the arithmetic if statement which we observed in code available to us. The main example being when two of the label references

```
      IF(FOO-MAX) 6,6,5
5     MAX=FOO
6     IF(FOO-MIN) 7,8,8
7     MIN=FOO
8     CONTINUE
```

Figure 3.6: Idiomatic use of arithmetic if statement

```
1.        if (foo > max) then
2.           go to 5
3.        else
4.           go to 6
5.        end if
6.     5  max ← foo
7.     6  if (foo < min) then
8.           go to 7
9.        else
10.          go to 8
11.       end if
12.    7  min ← foo
13.    8  continue
```

Figure 3.7: Arithmetic if statement transformed

are the same. Figure 3.6 shows such example. Using the semantics of the arithmetic if statement we can clearly see that in the first case, we have $foo - max <, =, > 0$. Since the first two labels are the same (6), and additionally the arithmetic expression is a subtraction we can easily create a more meaningful Boolean expression. In this particular case $foo > max$ is really what is meant by the underlying code. Whether to use $>$ or $\leq$ is difficult to tell by simply looking at the syntax tree. However we feel confident that using a boolean test inside an if statement instead of an arithmetic one is much more natural and exposes a more abstract meaning. We chose to go with $>$ for the first case and similarly $<$ for the second one. Thus this piece of code would be transformed to the printed Fortran-M code in Figure 3.7.

This form could be further improved using control flow analysis to guide the removal of the goto statements. However, as mentioned in Section 3.2.5, we decided that graph restructuring was beyond the scope of this work.

Another simple transformation occurs when label a and c are the same but b is different. In this case we transform the statement in a similar fashion to that previously described. The statement becomes an if-then-else where the Boolean expression is $expr = 0$.

3.2.4 Scoping Of Variables

So far, variables have been assumed to have a scope that spans the subprogram that contains them, or in the case of global variables, their common scope within the compilation unit (source file). Nevertheless, restricting the scope of variables to the only code block they are referenced in could be useful. By code block we mean

a sequence of statements with the same control flow parent. For example all the statements guarded by an if-statement are part of the same code block. The body of subprograms, if-thens, else-ifs, elses, do-loops, etc. are blocks. Restricting the scope of a variable to the particular if-branch it occurs in could for example expose a case of over-generalising in a subprogram where an if-then-else should really be two separate subprograms with fewer arguments. We have observed this practise to be common in such Fortran software as BLAS[14, 15, 24, 26] and LAPACK[2, 27].

Nevertheless, diving into this endeavour proved more difficult than we thought. The initial work yielded no immediate benefits. Therefore we decided to leave this task for future investigation.

3.2.5 Control Flow Analysis

Using control flow analysis to transform lower level control structures such as gotos into higher level structures like if-then-else and while loops is a desirable feature of this process. We set out to research various techniques to eliminate gotos from code. After investigating the techniques [4, 25] for implementing such treatment, we felt that the benefits did not warrant the effort required. Hence we left goto removing as future work. As we were mainly interested in transforming loop structures, we came across a method [16] that ignored the loop structures like do-loops and simply used the control flow graph directly to determine recurrence equations for looping control. This influenced our decision to use this method instead, described in Section 4.1.

Chapter 4

Linear Algebra Code Abstraction

The most ambitious goal of this project is to create a reverse engineering method that automatically abstracts algorithmic details from linear algebra software. Linear algebra involves vectors and matrices. At the implementation level, this means repeating an arithmetic operation inside one or more loops. Loop algorithms often hide their purpose at a low abstraction level. This is a consequence of the implementation programming language that does not allow for expressing mathematical abstractions, frequently due to performance reasons. As such, it is of great importance that our method be able to abstract loops into higher level linear algebra operations.

While attempting to recognise patterns in loops we came to the conclusion that simple regular expressions were not powerful enough. The pattern matching method, even for simple loops with no branches and a single statement, is too fragile to be useful in reverse engineering loops, as shown in example 2.4. For these reasons we opted for a more general method that would work for simple, as well as slightly more complex loops. This method consists of symbolically obtaining the recurrence relation for variables which state is changed inside the loop. If possible, we then attempt to solve the recurrence equations, and hopefully obtain the closed-forms. Hence we empower our pattern-based heuristic method with symbolic analysis.

4.1 Recognising Loop Structures

The method we chose for performing symbolic analysis is that of Fahringer and Scholz (F&S) [16]. This method employs the control flow graph instead of relying on the abstract syntax tree. This has the advantage that any cycle in the control flow graph, whether it be a do-loop or an if-goto loop, can be analysed as a loop. Despite this advantage, we decided to treat only do-loops due to difficulties in determining the closed-form of other loop forms. We must therefore begin by constructing a control

flow graph from the syntax tree. As F&S do not describe the creation of control flow graphs, we use the method outlined in [1] to create basic code blocks (vertices) and join them with control flow edges. This creates slightly different control flow graphs than F&S' graphs, as code blocks can contain multiple statements; for example, sequential assignments that don't have splits or joins in the control flow are part of a single block.

Once we obtained the graph, we label the edges either as forward or back edges. This is done using a simple depth-first search [10], where a back edge is one which joins the current node to a node already processed. Following the symbolic analysis algorithm, we extend the control flow graph by adding nodes that clearly identify the boundaries of each loop. These boundaries are the loop entrance, or pre-loop header (PH), the loop's end of body, or post-body (PB), and the loop exits, or post-exits (PE).

The algorithms to insert PH and PB are simple, and very similar: after detecting nodes with back-edges (loop header nodes), we add the new node to the graph, draw an edge from the new node to the loop header node, then replace all edges incoming to the loop header by edges incoming to the new node. Figure 4.1 shows the transition.

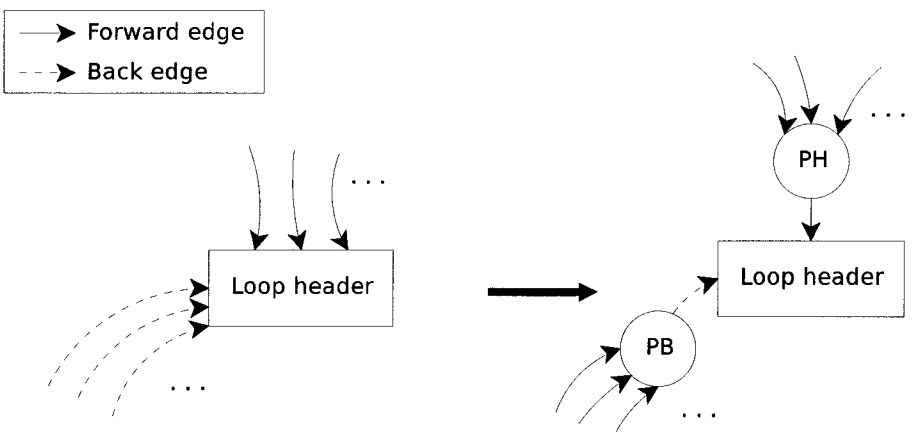


Figure 4.1: Adding pre-header and post-body nodes

Prior to detecting where PE nodes need to be added, the previous task of inserting PH and PB nodes must be completed. There is the obvious exit from a loop header node that is a branch. However to find exit edges from within the loop body we must have a better algorithm. Such algorithm requires that we ensure the loop subgraph is reducible, which is accomplished using a known test [21, 32]. Knowing that the control flow graph is reducible allows us to find where to add PE nodes with greater ease. These nodes are inserted by using a backwards breadth-first search starting at the PB node, marking the nodes when they are encountered. If a node has an

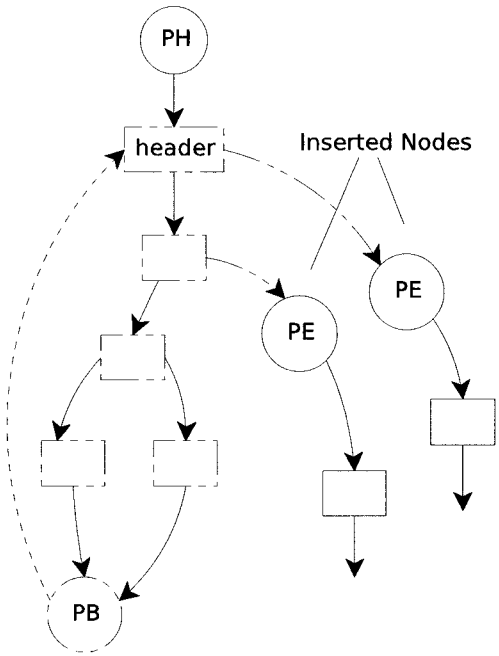


Figure 4.2: Adding post-exit nodes

outgoing forward edge to an unmarked node then we know this must be an exit. Thus we insert a node to mark an exit of the loop, as shown in Figure 4.2.

4.2 Symbolic Analysis Of Assignments To Scalars

The symbolic analysis of assignments to scalar variables differs significantly from that of assignments to arrays. The method outlined below in Sections 4.2.1 and 4.2.2 works only for assignment statements that redefine the value of a scalar variables. We cover the case of array assignments in Sections 4.3.1 and 4.3.2.

4.2.1 Finding Recurrence Equations

Once we obtain the extended control flow graph, we can begin the symbolic analysis. F&Suse the concept of *program contexts* to perform their symbolic analysis. A program context is a triple (s, t, p) where s is the current state of all variables in scope, t is the state condition and p is the path condition [16].

The state condition and path condition are used to perform the symbolic analysis of control flow branches. Our method does not use these as we determined that finding closed-forms of loops with internal branches or multiple exits was beyond the

intended scope of this work. Therefore, we only apply our method for loops with a straight-line code body.

To find the recurrence relation of the loop, we start by identifying the recurrence variables; these are variables that change value inside the loop body. We then include those variables into our state s, giving them an arbitrary recurrence value. We also search for initialisation assignment statements immediately prior to the loop header; those are recorded as the recurrence variables' value at time 0. And, since we only deal with do-loops, we initialise the loop variable to its initial value specified in the do-statement. For those variables for which we have no initial value we create a dummy initial value for the 0 recurrence value.

As opposed to F&S' algorithm which analyses entire control flow graphs, we only analyse loops. Moreover, we only analyse assignment statements since other types of statements may introduce complex side effects. Hence, we symbolically interpret the loops' body statements one by one to determine the recurrence equations for each loop. The flow graph nodes are visited in topological order [10], with the slight restriction that branches leading to post-exit nodes are taken first. This restriction is necessary to ensure that a loop's body is visited before its exit. The state value expressions of the recurrence variables are updated at each statement with new expressions if the state changes. Once we reach the PB node we have all the data required to create and attempt to solve the loop's recurrence equations.

To show more clearly how this works, we use as an example a simple dot product loop in Fortran, shown in Figure 4.3. We first determine that i and *stemp* are recurrence variables. These variables are replaced by sequences $i(k)$ and $stemp(k)$ respectively. Next we find that the initial value of i is 1; however the value of *stemp* at time 0 is unknown, and so we assign it an arbitrary symbolic value *stemp0*. This yields two states:

$$\begin{aligned} s_0 &= \{stemp(0) = stemp0, i(0) = 1\} \\ s_1 &= \{stemp(k+1) = stemp(k), i(k+1) = i(k)\} \end{aligned}$$

The state s_0 identifies the state of the variables prior to executing the loop. It is used after the symbolic analysis of the PB node to attempt to solve the recurrence equations.

```
    do 10 i = 1,n
      stemp = stemp + sx(i)*sy(i)
10  continue
```

Figure 4.3: Fortran vector dot product

The do-loop in Figure 4.3 contains two statements. The first one is the visible *stemp* assignment. The other is the implicit incrementation of the loop variable i, performed after all other statements of the loop body [3]. The symbolic interpretation of these two statements creates the final state values for the loop body:

$$\begin{aligned} stemp(k+1) &= stemp(k) + sx_{i(k)}\, sy_{i(k)} \\ i(k+1) &= i(k) + 1 \end{aligned}$$

4.2.2 Solving Recurrence Equations

Before we can solve the recurrence equations obtained above, we must create a data dependency graph. The importance of this graph comes into the process of solving recurrence equations. It allows us to solve multiple sets of recurrence equations in order of data dependency. This in turn yields solutions that are independent of each other, and thus can be executed in parallel. This is of utmost importance to enable removing the loop in the transformation we perform following the symbolic analysis.

For this graph, we define a data dependency (edge) between two variables (vertices) as a variable which is found in another variable's recurrence equation expression. The recurrence variable depends on the variables found in its expression. Non-recurrence variables are not included in the graph. This dependency graph should be a directed acyclic graph (DAG), if we exclude self-dependence. In cases where it is not we do not process the loop. Table 4.1 shows examples of acyclic and cyclic dependency graphs. This cyclic dependency example is somewhat special in the fact that it is a "coupled" recurrence system. In other words, this system consists of two recurrence systems that depend on each other. Maple is capable of handling some coupled system, such as this particular one. Nevertheless we decided against supporting any kind of cyclic dependency to restrict the scope of our work.

The process of solving recurrence equations for scalar variables is simplified by using the Maplesoft's Maple computer algebra system as a symbolic engine. We entrust this process to Maple's `rsolve` function and as such, the quality of our result, for better or worse, depends on it. We assume Maple's result to be correct, if a result can be obtained. If Maple is unable to solve the recurrence we simply stop processing the current loop.

Thus we continue our dot product example from Figure 4.3 by solving the recurrence equations. The dependency graph shows that *stemp* depends on i. Therefore we first find the solution to the recurrence equations of i:

$$i(k) = k + 1$$

	Acyclic	Cyclic

Code	`do 10 i=1,n`
	`  stemp=stemp+sx(ix)*sy(iy)`
	`  ix=ix+incx`
	`  iy=iy+incy`
	`10 continue`
		`do 10, i=2, n`
		`  v=u1`
		`  u1=-u0+2*x*u1`
		`  u0=v`
		`10 continue`

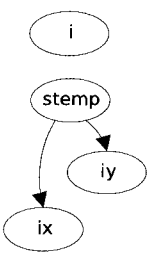
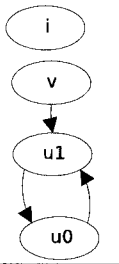
Graph		

Table 4.1: Examples of data dependency graphs

We can then substitute this solution into *stemp*'s recurrence equation,

$$stemp(k+1) = stemp(k) + sx_{k+1}\, sy_{k+1}$$

and solve it, expressing clearly that this is a dot product (see Section 4.5).

$$stemp(k) = stemp0 + \sum_{k0=1}^{k} (sx_{k0}\, sy_{k0})$$

The next step is to find the closed-forms of these solutions, and is explained in Section 4.4.

4.3 Symbolic Analysis Of Assignments To Arrays

In the case where a new value is assigned to an array element, the method outlined previously for creating a recurrence equation as for a scalar is impractical. For this reason we use a different technique for dealing with assignments to arrays.

4.3.1 Finding Recurrence Equations

By using the same technique naïvely on the loop in Figure 4.4 we would obtain the following recurrence equations:

$$\begin{aligned} sy_{i(k)}(k+1) &= sy_{i(k)}(k) + sa\, sx_{i(k)} \\ sy(0) &= sy0 \\ i(k+1) &= i(k)+1 \\ i(0) &= 1 \end{aligned}$$

However, the usual techniques for solving recurrences do not work on this set of

```
      do 10 i = 1,n
        sy(i) = sy(i) + sa*sx(i)
   10 continue
```

Figure 4.4: Fortran saxpy

equations. This is due to the fact that the recurrence is not only over time but over a set of indices. Following the footsteps of F&S we use an array overwriting operator $\oplus$ such that $A(k+1) = A(k) \oplus (A_j(k) \leftarrow e)$ redefines the array element at index j with the value of expression e. For example, `sy(i) = sy(i) + sa*sx(i)` in the context of Figure 4.4 has for recurrence relation:

$$sy(k+1) = sy(k) \oplus (sy_{i(k)}(k) \leftarrow sy_{i(k)}(k) + sa\, sx_{i(k)})$$

Additionally, to deal with any kind of loop and array assignment, F&S use a special operator $\rho(A(k), i)$ that enables reading from recurrence arrays. This operator takes into account the current overwritten state of an array, at any point. So in fact, the previous equation should be:

$$sy(k+1) = sy(k) \oplus (sy_{i(k)}(k) \leftarrow \rho(sy(k), i(k)) + sa\, sx_{i(k)})$$

4.3.2 Solving Recurrence Equations

As far as solving the recurrences of array assignments, we have found F&S' method to be very pessimistic in the sense that it assumes the worst possible form and combination of array assignments. This has the advantage of being very powerful in representing any kind of array recurrence. On the other hand it has the disadvantage of obfuscating the underlying meaning of a loop.

Since the latter reason is against a major goal of our project, we decided it would be best to use a less powerful, however more understandable representation for the

resulting solutions. Hence we chose partial functions as we thought they were a much more intuitive way of interpreting recurrences of array assignments.

To allow usage of partial functions we had to sacrifice the number of array assignment recurrences we could successfully determine. In addition to the cyclic data dependency restriction, as for scalar variables, we determined the following constraints were necessary:

1. The array shall appear in no other statements than its own assignment statement. This restriction excludes degenerate cases like this one:

```
      do 10, i=1, 5
        s = a(2,i) + s
        a(i,3) = a(i,3) + 1
10    continue
```

2. If the array is referenced in its assignment's right-hand side expression, the index expression shall be the same as in the left-hand side. This restriction excludes degenerate cases like this one:

```
      do 10, i=1, 5
        a(i) = a(3) + 1
10    continue
```

3. When dealing with assignments to matrices, at most one of the index expressions can contain recurrence variables. This enables us to treat matrices as vectors (see section 4.3.3).

We found through testing that these restrictions were sufficient to ensure the success of our method and are applicable in a large number cases. However it is quite possible that less constraining ones may yield more successful results with the same technique.

Once we have ensured that a loop containing assignment statements to arrays follows these conditions, we can attempt to solve the recurrence relation. One consequence of these conditions is that a symbolic expression will never contain more than one overwriting operator.

Continuing the `saxpy` example found in Section 4.3.1 we can use the result from the symbolic analysis to determine the solution to the recurrence. The first step is to replace $i(k)$ with its solution $k+1$ into the recurrence of array sy.

$$sy(k+1) = sy(k) \oplus (sy_{k+1}(k) \leftarrow \rho(sy(k), k+1) + sa\, sx_{k+1})$$

Further, we solve the recurrence by removing dependence on sequences of k on the right-hand side, shifting the recurrence index by minus one, embedding the iteration into the $\oplus$ operator, and rid the equation of the ρ operator. This solving can be done since the recurrence is essentially of degree zero and restrained by our rules. The one exception is when the index is constant with respect to the loop; in other words, that the index expression contains no recurrence variables. In this case we can be rid of the $\oplus$ and treat the assignment as scalar and use Maple's `rsolve` (see Section 4.3.3).

$$sy(k) = sy \bigoplus_{k0=1}^{k} (sy_{k0} \leftarrow sy_{k0} + sa\, sx_{k0})$$

The lower bound on the $\oplus$ operator index $k0$ is determined to be $i(0) = 1$, or more generally the recurrence of the whole index expression at time 0. Similarly, the upper bound is the index expression at time $k - 1$, in this case $i(k - 1) = k$.

This allows us to proceed representing the recurrence by using a partial function and a quantifier over all the integers. This quantification combined with the facts that in Fortran the size of an array is unknown, and that indices can be negative, ensures complete treatment of the loop.

$$\forall k0 \in \mathbb{Z} : sy_{k0} = \begin{cases} `sy_{k0} + sa\, sx_{k0} & \text{if } 1 \leq k0 \leq k \\ `sy_{k0} & \text{otherwise} \end{cases}$$

4.3.3 Arrays With Multiple Indices

With respect to symbolic analysis, there are three different ways indices can appear in recurrence arrays :

1. All index expressions are constant with respect to the loop; that is no index expression contains a recurrence variable (either vector or matrix). This case is very simple to handle. Instead of using $\oplus$ during the symbolic analysis of each statement, we treat the array reference as a scalar. Maple can handle the constant array element just as well as a scalar.

2. Only one index expression contains recurrence variables. This case was described in the previous section (4.3). We can have a vector with a non-constant index, or a matrix with one constant index and the other not. In the later case we treat the matrix as a vector.

3. Two indices contain recurrence variables (only matrices). As per our rules we do not handle this case. It involves using a vector of indices and the treatment is sizeably more difficult. However, our observations tell us this scenario happens

infrequently; it may occur when performing certain operations on symmetric matrices.

4.4 Finding Loop Closed Forms And Eliminating The Loops

Our ultimate goal in this exercise is to remove all the statements from the loop and let them stand by themselves as assignment statements; thus we eliminate the loop. In order for this transformation to be performed successfully we need to determine the number of times the loop will execute. Combined with the recurrence equation solutions in terms of iteration parameter k, we can find a statement that does not require a loop to obtain the same final value for the recurrence variables.

This task is greatly simplified in Fortran due to the semantics of the do-loop. Indeed all FORTRAN 77 references available to us [3, 11, 12] agree that the number of iterations, follows the semantics outlined below under all circumstances. The do-statement has the form,

```
DO s[,] v=e1, e2 [,e3]
```

where s is the label of the loop's terminal statement, *e1* and *e2* are respectively the initial and final value of the loop variable v, and *e3* is an optionally specified increment to v after each iteration. When *e3* is left unspecified, the increment is 1, and can never be 0. Given the above definition, the number of iterations is given by this expression:

$$\max\left(\left\lfloor \frac{e2 - e1 + e3}{e3} \right\rfloor, 0\right)$$

These semantics cannot be altered by assigning a value to the loop variable within the loop body. Not only is this forbidden and enforced at the syntax level (e.g. g77), but the semantics also specify that the exit condition of the loop is when an independent iteration counter reaches 0. The initial value of this counter is determined prior to the loop execution and decremented by one after each iteration. If the iteration counter's value is 0 the loop will not execute.

The number of iterations, for both our previous examples in Figures 4.3 and 4.4, are equal to n. By giving this value to k in our recurrence solutions we obtain the following results:

$$\begin{aligned} stemp(n) &= stemp0 + \sum_{k0=1}^{n} (sx_{k0}\, sy_{k0}) \\ i(n) &= n + 1 \end{aligned}$$

and

$$\begin{aligned} \forall k0 \in \mathbb{Z} : sy_{k0} &= \begin{cases} `sy_{k0} + sa\, sx_{k0} & \text{if } 1 \leq k0 \leq n \\ `sy_{k0} & \text{otherwise} \end{cases} \\ i(n) &= n + 1 \end{aligned}$$

respectively. The next step required to complete the transformation is to get rid of artifacts gathered during the recurrence solving, and create a proper assignment statement that can be part of the code. Therefore we remove recurrence arguments on the left-hand side, change the equalities to assignments, and return dummy initial values to the former variable, like this:

$$stemp \leftarrow stemp + \sum_{k0=1}^{n} (sx_{k0}\, sy_{k0})$$

For introduced variables like *k0* we can reliably assume that they are declared with a scope local to their sub-expression, and have a type inferred by their mathematical context, in this case a scalar integer.

At least we can remove the loop. Since all the recurrences were solved in order of data dependency, the result consists of parallel statement. Further, we must enclose this parallel block of assignment statements inside an if-statement. This guard condition is necessary to ensure nothing happens when the loop would not execute. This condition tests that the maximum number of iterations is greater than zero.

When the evaluation of the recurrence solution given the number of iterations fails, we assume that the solution is correct and still eliminate the loop. We simply leave the solution as is, with an additional evaluation clause specifying the value of k, the number of iterations. The example in Section 5.3.7 shows this transformation.

4.5 Linear Algebra Patterns

In this section we present a number of patterns specific to linear algebra that can be transformed by our method.

4.5.1 Dot Product

As we previously described in Section 4.2, a sum of two multiplied vector terms is a dot product. Therefore we would like to recognise this pattern and propose a transformation that preserves the semantics. More specifically, we cannot simply drop the range of indices of these vectors. Moreover, some dot products use vector

index increments not equal to 1, sometimes not even the same for both vectors. Our transformed form must take all these cases into account. The general pattern in Fortran-M is the following:

dotProduct : #(SUM #(MULT #(x:NAME ix:*expr*) #(y:NAME iy:*expr*))
#(EQ i:NAME #(RANGE a:*expr* b:*expr*))) ;

$$\sum_{i=a}^{b} x_{ix}\, y_{iy}$$

For the simple case, when $a = 1$, and $ix = iy = i$ we can use a simplified notation:

$$\vec{x}^{\,b} \cdot \vec{y}^{\,b}$$

Otherwise, if ix and iy are each a linear expression of i, we use a more general notation that defines a projection of each vector.

$$\vec{x}_{ix}^{\,a \le i \le b} \cdot \vec{y}_{iy}^{\,a \le i \le b}$$

4.5.2 Vector And Matrix Initialisation

Vector initialisation manifests itself in the form of a loop with an assignment to an array. For matrices, this is usually done in two nested loops. The general pattern for a vector is,

vectorInit : #(FORALL *expr* #(ASSIGN #(x:NAME ix:*expr*)
#(PIECEWISE c:*constant* #(AND #(LT a:*expr* i:*expr*)
#(LT i:*expr* m:*expr*)) *expr*))) ;

$$\forall i \in \mathbb{Z} : x_{ix} \leftarrow \begin{cases} c & \text{if } a \le i \le m \\ x_{ix} & \text{otherwise} \end{cases}$$

where c is constant value. For the simple case, as for the dot product, we use a less cumbersome notation:

$$\vec{x}^{\,b} \leftarrow c$$

For vectors with more complex index expressions, we display all the necessary information similarly to the vectors in a dot product. Matrices must display simple index

expressions for us to consider the transformation since, to restrict the scope, we did not wish to wade into matrix projection.

$$\vec{x}_{ix}^{\,a \le i \le m} \leftarrow c$$

For a matrix we must proceed in two iterations of symbolic analysis and pattern matching. The square brackets below denote a list of vectors, where the index gives the vector at the index' position in the list.

$$\forall i \in \mathbb{Z} : x_{i,j} \leftarrow \begin{cases} c & \text{if } 1 \le i \le m \\ x_{i,j} & \text{otherwise} \end{cases} \implies ([\vec{x}]_j)^m \leftarrow c$$

followed by:

$$\forall j \in \mathbb{Z} : ([\vec{x}]_j)^m \leftarrow \begin{cases} c & \text{if } b \le j \le n \\ ([\vec{x}]_j)^m & \text{otherwise} \end{cases} \implies \vec{x}^{m \times n} \leftarrow c$$

4.5.3 Vector Copy

The simple operation of copying a vector to another vector is an easy one to recognise. Its characteristic pattern is an assignment from a vector to a vector with no arithmetic operation. Our pattern expects that the index expression of both vectors is the same:

$$\forall i \in \mathbb{Z} : x_{ix} \leftarrow \begin{cases} y_{iy} & \text{if } a \le i \le m \\ x_{ix} & \text{otherwise} \end{cases}$$

When $a = 1$ and $ix = iy = i$ is a simple scalar variable, we obtain,

$$\vec{x}^{\,m} \leftarrow \vec{y}^{\,m}$$

otherwise we transform our pattern as such:

$$\vec{x}_{ix}^{\,a \le i \le m} \leftarrow \vec{y}_{iy}^{\,a \le i \le m}$$

4.5.4 Saxpy

The saxpy operation, an acronym for "(in **s**ingle precision floating point arithmetic) a constant (**a**) times a vector (**x**), **p**lus a vector (**y**)", was presented in Section 4.3.1.

Its pattern strikes a resemblance to the solution of the recurrence equation:

$$\forall i \in \mathbb{Z} : y_{iy} \leftarrow \begin{cases} y_{iy} + c\, x_{ix} & \text{if } a \leq i \leq m \\ y_{iy} & \text{otherwise} \end{cases}$$

Similarly to other vector operations, when $a = 1$ and $ix = iy = i$ is a simple scalar, we obtain the first assignment. For the more complicated cases we obtain the second.

$$\begin{aligned} \vec{y}^{\,m} &\leftarrow \vec{y}^{\,m} + c\, \vec{x}^{\,m} \\ \vec{y}_{iy}^{\,a \leq i \leq m} &\leftarrow \vec{y}_{iy}^{\,a \leq i \leq m} + c\, \vec{x}_{ix}^{\,a \leq i \leq m} \end{aligned}$$

4.5.5 Matrix Multiplication

We have already shown the transformation a matrix multiplication algorithm should undergo in Section 2.1.1. We now give more details on how our method handles the patterns, from the result of the symbolic analysis to the final transformation. Similarly to the patterns described in the previous section, it is important that no information, such as the sizes of the matrices, be lost. Like the matrix initialisation pattern, we decided to only recognise simple matrix multiplication to narrow the scope. As mentioned previously, in the first step, the innermost loop is analysed, and we find a dot product. Thus, our first rewriting is the following:

$$z_{i,j} \leftarrow z_{i,j} + \sum_{k=1}^{p} (x_{i,k}\, y_{k,j}) \implies z_{i,j} \leftarrow z_{i,j} + (([\vec{x}]_i)^T)^p \cdot ([\vec{y}]_j)^p$$

By transposing vector y we ensure that the j index is for column vectors and not rows. This assignment combined with the second inner loop delivers a vector operation after symbolic analysis:

$$\forall j \in \mathbb{Z} : z_{i,j} \leftarrow \begin{cases} z_{i,j} + (([\vec{x}]_i)^T)^p \cdot ([\vec{y}]_j)^p & \text{if } 1 \leq j \leq n \\ z_{i,j} & \text{otherwise} \end{cases}$$

This pattern yields:

$$(([\vec{z}]_i)^T)^n \leftarrow (([\vec{z}]_i)^T)^n + (([\vec{x}]_i)^T)^p \ \vec{y}^{\,p \times n}$$

Finally we symbolically analyse the last loop to obtain,

$$\forall i \in \mathbb{Z} : (([\vec{z}]_i)^T)^n \leftarrow \begin{cases} (([\vec{z}]_i)^T)^n + (([\vec{x}]_i)^T)^p \ \vec{y}^{\,p \times n} & \text{if } 1 \leq i \leq m \\ (([\vec{z}]_i)^T)^n & \text{otherwise} \end{cases}$$

which we transform to a much more appealing matrix multiplication:

$$\vec{z}^{\,m\times n} \leftarrow \vec{z}^{\,m\times n} + \vec{x}^{\,m\times p}\ \vec{y}^{\,p\times n}$$

Chapter 5

The Reverse Engineering Tool

5.1 Environment

The choice of using Java for the tools came about for many reasons. The first being our familiarity with the language. Some may argue rightfully that a functional programming language would have been better suited for the task. While we generally agree with the proposition, learning a new programming language combined with a completely different programming paradigm would have severely hampered our progress. Another reason was the ANTLR framework which offered a very powerful set of tools. ANTLR can also generate C++ or C#, but we decided to avoid the former for its dangerous low-level features, and the later for its lack of a mature environment for any operating system other than Microsoft Windows. Finally, since the tools from this project may be used by a variety of people inside or outside of academia, we decided to use a programming language which is familiar to a large majority of people, allowing them to use or maintain the tools. Java fits those criteria.

5.2 Parsing

In order to obtain an abstract syntax tree of the Fortran source code, it was decided to use a parser generator and Fortran grammar. The tool was chosen so that it would give us a flexible framework to both parse the language but also do tree pattern matching and transformations. After much deliberation we chose ANTLR[30]. ANTLR is an LALL(k) parser generator; that is a parser generator that creates recursive-descent parsers from non-left-recursive grammars—unlike the well known yacc LR parser generator—with an arbitrary fixed look-ahead (k). ANTLR has an EBNF-like syntax to create both parsers and lexical analysers, as well as a tree grammar syntax to parse abstract syntax trees. It moreover allows semantic actions to be associated with

grammar productions. These semantic actions can be specified in either ANTLR's own non-platform-specific language, which is very restrictive, or in the target language itself, in our case Java.

The ANTLR grammar used to create our parsing tool was ported from a PCCTS grammar (ANTLR's predecessor). That grammar was itself an adaptation of William Waite's grammar for Eli[33]. The PCCTS grammar was meant to be used in conjunction with f2c[17]'s lexer. Since we wanted to use Java for the tools we decided to create our own lexer. We also made several modifications to the PCCTS grammar which had incorporated Fortran 90 constructs which we do not want to use.

ANTLR does however have a few drawbacks. The main drawback with respect to Fortran is that it has difficulty handling languages with non-reserved keywords. In Fortran, identifiers can have the same name as Fortran keywords. An example of this is the "real" keyword for typing single-precision floating point numbers, and the intrinsic real() function that returns the real part of a complex number. As a result, programmers can use keywords of suitable length (less than seven characters) as identifiers. Thus "real" could be used as a variable name. This makes parsing much more difficult.

Another important drawback is that the lexer we created has some important caveats. The main one is that it does not respect the rule that in Fortran, blank characters (spaces) have no significance. For example you can have space inside variable names, or no space in statements such as this do loop `DO50I=1,5` with loop variable i and terminal statement at label 50, or this assignment `DO50I=1.5` of the real value 1.5 to variable `DO50I`. This is something that is very difficult to perform in a standalone lexer as it requires knowledge about the language grammar; a concept known as stateful lexing. To simplify our task it was therefore decided to create a lexer that assumed all tokens would contain no blank characters, and that spaces will be used to delimit tokens like keywords, symbolic names, labels and constant values. This decision has not proved an issue in the code available to us.

Despite its drawbacks, ANTLR is a great tool in practise. A noteworthy feature of the lexer is that it does not discard comments. Since our goal is to reverse engineer the code for better human comprehension, up to design or even requirements level, it should be obvious that discarding code comments could be a harmful loss of information. Hence comments are kept as part of the syntax tree since they contain valuable information which can complement a piece of code with meaning that no automatic tool can reverse engineer.

Lastly, the most compelling reason for using ANTLR is its automatic generation of abstract syntax trees. Since we wanted Fortran-M to be based on a Fortran syntax tree this feature has proved very handy. The syntax tree creation from the grammatical rules can be customised and is very flexible.

5.3 Sample Results

The following sections show several example of input given to our proof-of-concept tool and the output obtained.

The output produced by our tool should be mostly straightforward to understand. In any case, we provide simple explanations for it. Each line of code is numbered in the left-most column. The second left-most column is for code labels (numbers), and to mark the beginning of a subprogram with a "★". Fortran-M **keywords** , comments and *variables* are all all presented with their own font styles, while subprogram names retain a normal upright serif font. Assignments are denoted by a "←," and expressions should follow usual mathematical notation, such as "∧" indicating logical-and, "∀" denoting a universal quatification, "!" indicating a factorial term, and so forth. As stated in Section 3.2.1, assignment statements demarked with a double parallel bar indicates that these statements can be executed in parallel. The subprogram parameters are shown in their proper order on the subprogram's declaration statement. Following Fortran conventions, the symbols are typed using type statements in the subprogram's body. Declaration of input, output and update parameters, local and global variable as well as external and intrinsic routines are clearly identified and grouped using comments.

5.3.1 Dot Product

This example is taken from LAPACK BLAS [24, 26]. It is used to calculate the dot product of two vectors, $\vec{sx}$ and $\vec{sy}$. From the result we notice that the dot product is performed in three different places. Taking a closer look at the code we see that in the first case, the dot product is calculated such that the increment to the vector index is not 1. This creates a projection of each vectors represented within the notation. In the second case we have a simple dot product without frills. However, it can be deduced from the code that the second and third case are complementary. That is, the code uses an optimisation technique known as "loop unrolling." The vectors are partitioned in section of five elements. The second dot product performs the remainder while the third calculates the whole sections in increments of five. Despite this optimisation our method is able to determine that the code simply performs a sequence of dot products that it then adds to form the final result.

```
      real function sdot(n,sx,incx,sy,incy)
c
c     forms the dot product of two vectors.
c     uses unrolled loops for increments equal to one.
```

```
c     jack dongarra, linpack, 3/11/78.
c     modified 12/3/93, array(1) declarations changed to array(*)
c
      real sx(*),sy(*),stemp
      integer i,incx,incy,ix,iy,m,mp1,n
c
      stemp = 0.0e0
      sdot = 0.0e0
      if(n.le.0)return
      if(incx.eq.1.and.incy.eq.1)go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if(incx.lt.0)ix = (-n+1)*incx + 1
      if(incy.lt.0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        stemp = stemp + sx(ix)*sy(iy)
        ix = ix + incx
        iy = iy + incy
   10 continue
      sdot = stemp
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n,5)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        stemp = stemp + sx(i)*sy(i)
   30 continue
      if( n .lt. 5 ) go to 60
   40 mp1 = m + 1
      do 50 i = mp1,n,5
        stemp = stemp + sx(i)*sy(i) + sx(i + 1)*sy(i + 1) +
```

```
     *   sx(i + 2)*sy(i + 2) + sx(i + 3)*sy(i + 3) + sx(i + 4)*sy(i + 4)
   50 continue
   60 sdot = stemp
      return
      end
```

1. ★ **subprogram** sdot($n, sx, incx, sy, incy$)
2.
3. Function output:
4. **real** $sdot = 0.0e0$
5.
6. Input parameters:
7. **real** $sy(*)$
8. **integer** $incy$
9. **real** $sx(*)$
10. **integer** n
11. **integer** $incx$
12.
13. Intrinsic functions:
14. **intrinsic** mod
15.
16. Local variables:
17. **integer** ix
18. **integer** i
19. **integer** $mp1$
20. **integer** m
21. **real** $stemp = 0.0e0$
22. **integer** iy
23.
24. Executable statements:
25.
26. forms the dot product of two vectors.
27. uses unrolled loops for increments equal to one.
28. jack dongarra, linpack, 3/11/78.
29. modified 12/3/93, array(1) declarations changed to array(*)
30.
31. **if** $(n \leq 0)$ **then**
32. **return**
33. **end if**

```
34.       if (incx = 1 ∧ incy = 1) then
35.           go to 20
36.       end if
37.
38.       code for unequal increments or equal increments
39.       not equal to 1
40.
41.       ||  ix ← 1
          ||  iy ← 1
42.       if (incx < 0) then
43.           ix ← (−n + 1) incx + 1
44.       end if
45.       if (incy < 0) then
46.           iy ← (−n + 1) incy + 1
47.       end if
48.       if (n > 0) then
              ||  stemp ← stemp+ sx⃗ 1≤k0≤n / ix+incx (k0−1) · sy⃗ 1≤k0≤n / iy+incy (k0−1)
49.           ||  ix ← ix + incx n
              ||  iy ← iy + incy n
              ||  i ← n + 1
50.       end if
51.       sdot ← stemp
52.       return
53.
54.       code for both increments equal to 1
55.       clean-up loop
56.
57.   20  m ← mod(n, 5)
58.       if (m = 0) then
59.           go to 40
60.       end if
61.       if (m > 0) then
              ||  stemp ← stemp+ sx⃗ m · sy⃗ m
62.           ||  i ← m + 1
63.       end if
64.       if (n < 5) then
65.           go to 60
66.       end if
67.   40  mp1 ← m + 1
```

68. **if** $(\text{int}(\frac{n+5-mp1}{5}) > 0)$ **then**

69. $stemp \leftarrow stemp + \vec{sx}_{mp1+5\,k0-5}^{\,1 \le k0 \le \frac{n+5-mp1}{5}} \cdot \vec{sy}_{mp1+5\,k0-5}^{\,1 \le k0 \le \frac{n+5-mp1}{5}}$
$+ \vec{sx}_{mp1+5\,k0-3}^{\,1 \le k0 \le \frac{n+5-mp1}{5}} \cdot \vec{sy}_{mp1+5\,k0-3}^{\,1 \le k0 \le \frac{n+5-mp1}{5}}$
$+ \vec{sx}_{mp1+5\,k0-2}^{\,1 \le k0 \le \frac{n+5-mp1}{5}} \cdot \vec{sy}_{mp1+5\,k0-2}^{\,1 \le k0 \le \frac{n+5-mp1}{5}}$
$+ \vec{sx}_{mp1+5\,k0-1}^{\,1 \le k0 \le \frac{n+5-mp1}{5}} \cdot \vec{sy}_{mp1+5\,k0-1}^{\,1 \le k0 \le \frac{n+5-mp1}{5}}$
$+ \vec{sx}_{mp1+5\,k0-4}^{\,1 \le k0 \le \frac{n+5-mp1}{5}} \cdot \vec{sy}_{mp1+5\,k0-4}^{\,1 \le k0 \le \frac{n+5-mp1}{5}}$

$i \leftarrow n+5$

70. **end if**
71. 60 $sdot \leftarrow stemp$
72. **return**
73. **end**

5.3.2 Saxpy

This routine is taken from LAPACK BLAS [24, 26]. It is used to add two vectors, one of which can be multiplied by a constant. As for `sdot` there are three cases where we find the vector operation: the first one uses a projection of the vectors, the second deals with the remainder of the unrolled loop vector elements, and the third deals with chunks of vectors in an unrolled loop. The results for the first and second are those to be expected. For the third case we note however that the loop was left untouched, without being transformed. This is due to not complying with our restrictions on the form of array assignments.

```
      subroutine saxpy(n,sa,sx,incx,sy,incy)
c
c     constant times a vector plus a vector.
c     uses unrolled loop for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c     modified 12/3/93, array(1) declarations changed to array(*)
c
      real sx(*),sy(*),sa
      integer i,incx,incy,ix,iy,m,mp1,n
c
      if(n.le.0)return
      if (sa .eq. 0.0) return
      if(incx.eq.1.and.incy.eq.1)go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if(incx.lt.0)ix = (-n+1)*incx + 1
      if(incy.lt.0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        sy(iy) = sy(iy) + sa*sx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c
c        code for both increments equal to 1
```

```
c
c
c        clean-up loop
c
   20 m = mod(n,4)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        sy(i) = sy(i) + sa*sx(i)
   30 continue
      if( n .lt. 4 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,4
        sy(i) = sy(i) + sa*sx(i)
        sy(i + 1) = sy(i + 1) + sa*sx(i + 1)
        sy(i + 2) = sy(i + 2) + sa*sx(i + 2)
        sy(i + 3) = sy(i + 3) + sa*sx(i + 3)
   50 continue
      return
      end
```

1. ★ **subprogram** saxpy($n, sa, sx, incx, sy, incy$)
2.
3. Input parameters:
4. **integer** *incy*
5. **real** *sa*
6. **real** $sx(*)$
7. **integer** n
8. **integer** *incx*
9.
10. Output parameters:
11. **real** $sy(*)$
12.
13. Intrinsic functions:
14. **intrinsic** mod
15.
16. Local variables:
17. **integer** *ix*
18. **integer** i
19. **integer** *mp1*

```
20.     integer m
21.     integer iy
22.
23.     Executable statements:
24.
25.     constant times a vector plus a vector.
26.     uses unrolled loop for increments equal to one.
27.     jack dongarra, linpack, 3/11/78.
28.     modified 12/3/93, array(1) declarations changed to array(*)
29.
30.     if (n ≤ 0) then
31.         return
32.     end if
33.     if (sa = 0.0) then
34.         return
35.     end if
36.     if (incx = 1 ∧ incy = 1) then
37.         go to 20
38.     end if
39.
40.      code for unequal increments or equal increments
41.      not equal to 1
42.
43.     ix ← 1
        iy ← 1
44.     if (incx < 0) then
45.         ix ← (−n + 1) incx + 1
46.     end if
47.     if (incy < 0) then
48.         iy ← (−n + 1) incy + 1
49.     end if
50.     if (n > 0) then
51.         sy[1≤k0≤n, iy+incy(k0−1)] ← sy[1≤k0≤n, iy+incy(k0−1)] + sa sx[1≤k0≤n, ix+incx(k0−1)]
            ix ← ix + incx n
            iy ← iy + incy n
            i ← n + 1
52.     end if
53.     return
54.
55.      code for both increments equal to 1
```

```
56.          clean-up loop
57.
58.    20    m ← mod(n, 4)
59.          if (m = 0) then
60.              go to 40
61.          end if
62.          if (m > 0) then
63.              || sy^m ← sy^m + sa sx^m
                 || i ← m + 1
64.          end if
65.          if (n < 4) then
66.              return
67.          end if
68.    40    mp1 ← m + 1
69.          do 50, i = mp1, n
70.              sy_i ← sy_i + sa sx_i
71.              sy_{i+1} ← sy_{i+1} + sa sx_{i+1}
72.              sy_{i+2} ← sy_{i+2} + sa sx_{i+2}
73.              sy_{i+3} ← sy_{i+3} + sa sx_{i+3}
74.    50        continue
75.
76.          return
77.          end
```

5.3.3 Matrix Multiplication

The following example shows the result of applying our process to a simple matrix multiplication. In order to completely process the three loops we had to "cheat" our way through the symbolic analysis. This is due to the following reasons: our process is iterative and we process each loop on different iterations. The innermost loop is analysed first, then eliminated and transformed using vector patterns. This creates a dot product enclosed in an if-statement. As stated in Section 4.2.1 however, we only symbolically analyse loops which contain straight-line code.

To circumvent this issue, we use the knowledge that a loop with only a single if-statement in its body can be analysed if and only if the branch condition is static with respect to the loop; that is, it does not depend on any of the recurrence variables. Thus, for each iteration, the if-statement either always executes or never executes. This allows us to combine the if-statement's condition with the do-loop's entry condition.

Nevertheless we encounter a slight discrepancy. All the statements are enclosed in a single if-statement. There should be three if-statements that separate the execution of the assignment statements for each of the loop variables.

```
      subroutine mmult1(x, y, z, m, n, p)

      integer i, j, k
      integer m, n, p
      integer x(m, p), y(p, n), z(m, n)

      do 10 i=1, m
         do 20 j=1, n
            do 30 k=1, p
               z(i,j) = z(i,j) + x(i,k) * y(k,j)
30          continue
20       continue
10    continue

      end
```

1. ★ **subprogram** mmult1(x, y, z, m, n, p)
2.
3. Input parameters:
4. **integer** m

```
5.      integer y(p, n)
6.      integer p
7.      integer n
8.      integer x(m, p)
9.
10.     Updated parameters:
11.     integer z(m, n)
12.
13.     Local variables:
14.     integer i
15.     integer k
16.     integer j
17.
18.     Executable statements:
19.     if (0 < p ∧ 0 < n ∧ 0 < m) then
20.         || z⃗^{m×n} ← z⃗^{m×n} + x⃗^{m×p} y⃗^{p×n}
            || k ← p + 1
            || j ← n + 1
            || i ← m + 1
21.     end if
22.     end
```

5.3.4 Sigprod

This example calculates the sum of incrementally smaller factorial fractions. This fact is made much clearer by looking at the closed-form of w which is $1/n!$. The Γ function with two arguments here is defined by Maple as a hypergeometric function, the mathematical representation of which is beyond the scope of our work.

```
      integer function sigprod(n)
      integer n, i
      double precision s, w
      s = 1.0
      w = 1.0
      do 10, i=1, n
         w = w / i
         s = s + w
10    continue
      sigprod = s
      end
```

1. ★ **subprogram** sigprod(n)
2.
3. Function output:
4. **integer** *sigprod*
5.
6. Input parameters:
7. **integer** n
8.
9. Local variables:
10. **integer** i
11. **double precision** $w = 1.0$
12. **double precision** $s = 1.0$
13.
14. Executable statements:
15. **if** $(n > 0)$ **then**
16. $$\begin{aligned} w &\leftarrow \frac{1}{n!} \\ s &\leftarrow \frac{\mathrm{e}^1 \Gamma(n+1,1)}{n!} \\ i &\leftarrow n+1 \end{aligned}$$

```
17.     end if
18.     sigprod ← s
19.     end
```

5.3.5 FF2

The following function calculates the same value as `sigprod` in the previous section using a slightly different algorithm. However, the result varies slightly from that of `sigprod`. The reason for this marginal difference stems from the initial value given to the accumulators, s and *sum1* respectively. In `sigprod` the initial value of s is 1, while in this case $sum1 = 0$. If we take this difference into account, the result is in fact the same.

```
      double precision function ff2(n)
      integer n, i, fact
      double precision sum1
      sum1 = 0.0
      fact = 1.0
      do 60, i=1, n
         fact = fact * i
         sum1 = sum1 + 1/fact
60    continue
      ff2 = sum1
      end
```

1. ★ **subprogram** ff2(n)
2.
3. Function output:
4. **double precision** *ff2*
5.
6. Input parameters:
7. **integer** n
8.
9. Local variables:
10. **integer** i
11. **double precision** $sum1 = 0.0$
12. **integer** $fact = 1.0$
13.
14. Executable statements:
15. **if** $(n > 0)$ **then**
16. $$\left\| \begin{array}{l} fact \leftarrow n! \\ sum1 \leftarrow \dfrac{\mathrm{e}^1\,\Gamma(n+1,1) - n!}{n!} \\ i \leftarrow n+1 \end{array} \right.$$

```
17.     end if
18.     ff2 ← sum1
19.     end
```

5.3.6 Sigsum

This example is very much like `sigprod`. Instead of summing fractions however, the `sigsum` function sums the factorials. The Ei() function is defined by Maple as the exponential integral, the mathematical representation of which is beyond the scope of our work.

```
      double precision function sigsum(n)
      integer n, i
      double precision s, w
      s = 1.0
      w = 1.0
      do 20, i=1, n
         w = w * i
         s = s + w
20    continue
      sigsum = s
      end
```

1. ★ **subprogram** sigsum(n)
2.
3. Function output:
4. **double precision** *sigsum*
5.
6. Input parameters:
7. **integer** n
8.
9. Local variables:
10. **integer** i
11. **double precision** $w = 1.0$
12. **double precision** $s = 1.0$
13.
14. Executable statements:
15. **if** $(n > 0)$ **then**
16. $w \leftarrow n!$
    $s \leftarrow 1 - (\mathrm{e}^{-1}\,\mathrm{Ei}(2,-1) + (n+1)!\,\mathrm{e}^{-1}\,\Gamma(-n+1,-1)\,(-1)^n)$
    $i \leftarrow n+1$
17. **end if**
18. *sigsum* $\leftarrow s$
19. **end**

5.3.7 Sigsum1

This function should calculate the same value as `sigsum`. However we note some significant differences: Maple is unable to evaluate the recurrence solution with $k = n - 1$, as this results in an undefined value for $\Gamma(0)$. In this case, as prescribed in our method for finding closed-forms, in Section 4.4, we leave the expression unevaluated and display the value k should take. Dr. Jacques Carette (private correspondence) confirmed this result is correct however, since it is possible to obtain the result by finding the value of the sum and, by using limits, forcing $lim_{x \to 0}\Gamma(x) = \infty$. This in turns leads to terms becoming $1/\infty$ and being removed from the expression by simplification. The knowledge of Maple required to perform such operations however is far beyond the scope of this work.

```
      double precision function sigsum1(n)
      integer n, i
      double precision s
      s=n
      do 30, i=n-1, 1, -1
         s = i * (1 + s)
30    continue
      sigsum = s+1
      end
```

1. ★ **subprogram** sigsum1(n)
2.
3. Function output:
4. **double precision** *sigsum1*
5.
6. Input parameters:
7. **integer** n
8.
9. Local variables:
10. **integer** i
11. **double precision** s
12.
13. Executable statements:
14. $s \leftarrow n$
15. **if** $(n - 1 > 0)$ **then**

16. $$s \leftarrow \frac{(-1)^k\,\Gamma(k+1-n)\,(\Gamma(1-n)\,(\sum_{k1=0}^{k-1}\frac{(-1)^{k1}}{\Gamma(k1+1-n)})+s)}{\Gamma(1-n)},$$
    where $k = \max(0, n-1)$
    $i \leftarrow 0$
17. **end if**
18. *sigsum* $\leftarrow s+1$
19. **end**

5.3.8 Bessel

This example shows code that computes a Bessel function of the first kind. Although the results look much more complex than they need to be, we trust that the results from Maple are indeed correct for the given code. The BesselJ function is defined by Maple as a bessel function of the first kind, while the LommelS1 function is defined in terms of a hypergeometric function, the mathematical representation is beyond the scope of our work.

```
      double precision function bessel(z, nu, m)
      integer i, z, nu, m
      double precision res, u
      res = 0
      u = 1
      do 40, i=0, m-1
         res = res + u
         u = -u*z**2 / (4*(i+nu+1)*(i+1))
40    continue
      bessel = res
      end
```

1. ★ **subprogram** bessel(z, ν, m)
2.
3. Function output:
4. **double precision** *bessel*
5.
6. Input parameters:
7. **integer** z
8. **integer** m
9. **integer** ν
10.
11. Local variables:
12. **integer** i
13. **double precision** $u = 1$
14. **double precision** $res = 0$
15.
16. Executable statements:
17. **if** $(m > 0)$ **then**

18.
$$
\begin{aligned}
res \;&\leftarrow\; ((\mathrm{BesselJ}(\nu, z)\, 2^{\nu}\, z^{-\nu}\, z^2\, (m+\nu)!\, m! \\
&\quad +(-1)^m\, 4^{-m}\, (z^2)^m\, z^{2-(2\,m+\nu)}\, \mathrm{LommelS1}(2\,m+\nu+1, \nu, z) \\
&\quad -4^{-m}\, (-1)^m\, (z^2)^m\, z^2)\, \nu!) \frac{1}{z^2\, (m+\nu)!\, m!} \\
u \;&\leftarrow\; \frac{(-1)^m\, 4^{-m}\, (z^2)^m\, \nu!}{(m+\nu)!\, m!} \\
i \;&\leftarrow\; m
\end{aligned}
$$

19. **end if**
20. *bessel* $\leftarrow$ *res*
21. **end**

5.3.9 Mystery

This function is not such a mystery, despite its misleading name. The calculations done inside the loop body are in fact identical to `bessel`. The difference is the number of iterations $(m + 1)$ as opposed to only m for `bessel`. Assuming that `bessel` is correct, the results show an off-by-one error that is more difficult to notice by simply looking at the code.

```
      double precision function mystery(z, nu, m)
      integer z, nu, m, i
      double precision res, t
      res = 0
      t = 1
      do 70, i=0, m
         res = res + t
         t = -t*z**2 / (4*(i+nu+1)*(i+1))
70    continue
      mystery = res
      end
```

1. ★ **subprogram** mystery(z, ν, m)
2.
3. Function output:
4. **double precision** *mystery*
5.
6. Input parameters:
7. **integer** z
8. **integer** m
9. **integer** ν
10.
11. Local variables:
12. **integer** i
13. **double precision** $t = 1$
14. **double precision** $res = 0$
15.
16. Executable statements:
17. **if** $(m + 1 > 0)$ **then**

$$
\begin{aligned}
res \leftarrow\ & ((\mathrm{BesselJ}(\nu, z)\, 2^{\nu}\, z^{-\nu}\, z^{2}\, (m+1)!\, (m+1+\nu)! \\
& +(-1)^{m}\, 2^{-2\,m+2}\, (z^{2})^{m}\, z^{4} \\
& +(-1)^{m+1}\, 2^{-2\,m+2}\, (z^{2})^{m}\, z^{2-(2\,m+\nu)}\, \mathrm{LommelS1}(2\,m+3+\nu, \nu, z))\, \nu!) \\
& \frac{1}{z^{2}\, (m+1)!\, (m+1+\nu)!}
\end{aligned}
$$

$$t \leftarrow \frac{(-1)^{m+1}\, 2^{-2\,m+2}\, (z^{2})^{m}\, z^{2}\, \nu!}{(m+1+\nu)!\, (m+1)!}$$

$$i \leftarrow m+1$$

end if

mystery ← *res*

end

5.3.10 Finite Element Analysis

We tested our tool against real scientific computation code. The program tested, provided by Dr. Dieter Stolle from the McMaster University Civil Engineering department, consists of four files (or modules) used in civil engineering to perform finite element analysis (stress/strain calculations). The modules have a combined 63 subprograms for a total of over 2000 lines of code. After four iterations of symbolic analysis and pattern matching, we recorded a total of 60 loops that could be and were completely analysed symbolically. However 115 loops could not be analysed, mostly due to containing call, read, write or if-statements, with a few containing array assignments that did not fall under our rules. This gives us a good approximation of our success rate, roughly 34.4%. As far as transformations were concerned, Table 5.10 shows the numbers for each patterns.

Type of pattern	*Successes*
Variable initialisations	6
Parallel assignments	80
Arithmetic if-statements	2
Dot products	20
Vector assignments	34

Table 5.10: Statistics of transformations

An interesting fact that stems from these figures is that out of 60 analysed loops, 54 (90%) of them were either dot products or vector assignments. This could be due to the fact that we target those transformations specifically. It also shows the vast potential of our method with software containing large amounts of scientific computation code. Another interesting fact is the relatively large number of parallel assignments found, which shows that with very primitive data flow analysis we were able to achieve a significant amount of parallelisation.

Most of the dot products appeared to have been part of a matrix multiplications. However, none of the matrix multiplications were completely transformed since all of them used temporary variables. The extra assignment statements prevented a match to our pattern: only having a single static if-statement inside the do-loop. Some subprograms are used only to initialise vectors and matrices with 0 values. As per our method in Section 4.5.2, both vector and matrix initalisations were recognised and transformed.

We encountered some issues with the result in certain cases of vector assignments. The transformation performed lead to having the wrong vector dimension or projection. We are positive this is due to our quatified array assignment pattern being too generic. A stronger condition would ensure a safer transformation, or more likely

leave it as is.

5.3.11 LAPACK's BLAS

We also tested our tool on LAPACK's basic linear algebra subprograms (BLAS) library [24]. However, since it contains duplication of most routines for different input data types (real, double precision, complex and double complex), we only ran the tool on the set of subprograms for data of type real. That is we only looked at files starting with "s", for single-precision.

BLAS contains 141 files—and an equal number of subprograms—33 of which are for data of type real. These 33 routines total over 6700 lines of code. A grep for do-loops on the source files showed a total of 367 loops. Out of these, 128 were successfully transformed, that is 34.9% success. Table 5.11 gives a summary of the transformations performed. Successes and issues similar to those of Section 5.3.10 were observed.

Type of pattern	*Successes*
Variable initialisations	22
Parallel assignments	215
Arithmetic if-statements	0
Dot products	33
Vector assignments	100

Table 5.11: Statistics of transformations on BLAS

Chapter 6

Concluding Remarks

We have had success with several aspects of this project. We can infer useful information, such as type, input or output status, of identifiers. This should not be surprising however since all Fortran compilers do most of this work. We perform some structural transformation such as improving the understanding of arithmetic if-statements, and parallelising statements that were previously thought to be sequential. The heaviest reverse engineering work is performed with symbolic analysis. This technique provides a robust way to determine the high-level meaning underlying deep within loops. We were successful with most do-loops containing only assignments of arithmetic expressions to scalar, as well as with a number of loops with assignments to arrays. Symbolic analysis enables us to reduce the number of patterns required to determining abstract function of loops. Building on this work we have multiple patterns to recognise linear algebra operations on vectors and matrices, transforming the operations to a higher-level of abstraction.

Our decision to transform the Fortran abstract syntax tree into Fortran-M has had a definitely positive outcome. The addition of mathematical operators to extend Fortran really was fruitful. This made Fortran-M especially flexible, giving way to a plethora of opportunities. Moreover, keeping the language tree-based made it particularly easy to work with. Using tree parsing to do everything from pattern matching to tree walking, made this possible.

6.1 Future Work

Symbolic analysis has proved to be a very promising technique for automated reverse engineering. It should be taken much further than what we have accomplished with it. More specifically, we would like to see the result of symbolic analysis of entire subprograms. This would allow, among other features, for small and simple subprograms

to be in-lined within their call, improving the understanding of the calling program. We believe this would be particularly true of Fortran since names are restricted to six characters. This feature often leads programmers to choose cryptic acronyms for naming subprograms. To fully analyse subprograms we would need to support many more kinds of statements. The most important one, and likely the most difficult, would be logical branches. It should also be possible to further some aspects of symbolic analysis that we currently perform. It should be possible to treat "coupled" systems: that is two recurrence variables that depend on each other. Maple's `rsolve` is able to handle such systems. Furthermore, relaxing some of our restrictions on array assignments we expect would lead to a greater number of solved recurrences. Finally, ensuring that error information is kept throughout symbolic analysis and transformations would be an interesting and quite possibly very rewarding action to take.

Control graph restructuring would bring many major improvements: to enhance the logical meaning of branches by creating if-then-else blocks and while loops. Gotos are famous for obfuscating the purpose of code [13], and removing them should improve the understanding of the code. We have also encountered several instances of over-generalisation of subprograms: a subprogram with many of its input parameters not being used in all code paths, for example in a body that is one large if-then-else. Combining goto-elimination with more flexible scoping of variable would permit us to differentiate code blocks and possibly separate them into more specialised subprograms. Not having gotos would, additionally, significantly ease the process of producing tabular expressions from the code. As tables are easy to read and understand, outputting them is a very desireable feature.

We would definitely like to see the outcome of using more powerful and thorough data flow analysis. Detecting more parallel compositions would surely follow for such work. More importantly, we believe that resulting from symbolic analysis, several local variables would be left un-read within a subprogram. Data flow analysis would permit us to remove them from the syntax tree.

It could also be interesting to see how far the output system could be taken. We believe our prototype has the potential to be a successful re-engineering tool. Enhancing the generation of Fortran code would likely be the priority. Nevertheless we would like to see output in C that could rival that of `f2c`, especially in understandability of the code. Possibly other high-level imperative languages such newer versions of Fortran, and Java which is gaining popularity amongst the scientific community. More extensive symbolic analysis may even yield surprising results translating to functional languages.

Appendix A

Fortran-M Details

A.1 Tree Notation in EBNF

languagedef = *rule*, { *rule* } ;
rule = *literal*, ':', *definitionlist*, ';' ;
definitionlist = *definition*, { '|', *definitionlist* } | '(', *definitionlist*, ')', (| '?' | '*' | '+') ;
definition = *node* | *tree* ;
tree = '#(', *node*, *tree*, {*tree*}, ')' ;
node = [*literal*, ':'], (*terminal* | '"', *keyword*, '"' | '.') ;
terminal = (* see list of terminal symbols in Appendix A.3 *) ;
keyword = (* see list of terminal symbols in Appendix A.3 *) ;
literal = *letter*, { *character* } ;
character = *letter* | *number* ;
letter = 'a' | 'b' | 'c' | ... | 'z' ;
number = '0' | '1' | '2' | ... | '9' ;

A.2 Language Grammar In Tree Notation

This grammar is by no means complete. Nevertheless it contains the majority of statements recognised and used by our system.

unit : #(CODEROOT (*subprogram*)+) ;
subprogram : #(SUBPROGRAM *codeblock*) | *comment* ;
label : (LABEL |) ;

codeblock : (*statement*)* ;
statement : *comment* | *parallel* | *assignment* | *ifStatement* | *doStatement* | *gotoStmt* | *returnStmt* | *continueStmt* | *callStmt* | *equivalenceStmt* ;
parallel : #(PARALLEL (*assignment*)+) ;
comment : (COMMENT)+ ;
assignment : #(EQUALS *label varRef expr*) ;
ifStatement : #("if" *label expr thenBlock* (*elseIfBlock*)* (*elseBlock*)?) ;
thenBlock : #(THENBLOCK *codeblock*) ;
elseIfBlock : #(ELSEIF *expr thenBlock*) ;
elseBlock : #(ELSEBLOCK *codeblock*) ;
doStatement : #("do" *label* LABELREF NAME *expr expr* (*expr*)? #(DOBLOCK *codeblock*)) ;
gotoStmt : #("go" *label* LABELREF) ;
returnStmt : #("return" *label*) ;
continueStmt : #("continue" *label*) ;
callStmt : #("call" *label externalFunction*) ;
dataStatement : #("data" (*dataStatementEntity*)+) ;
dataStatementItem : *varRef* | *dataImpliedDo* ;
dataStatementMultiple : (#(STAR (ICON | NAME) (*constant* | NAME)) | (*constant* | NAME)) ;
dataStatementEntity : *dse1 dse2* ;
dse1 : #(DIV (*dataStatementItem*)+) ;
dse2 : #(DIV (*dataStatementMultiple*)+) ;
dataImpliedDo : #(LPAREN *dataImpliedDoList* #(EQUALS NAME *expr expr* (*expr*)?)) ;
dataImpliedDoList : (*dataImpliedDoListWhat*)+ ;
dataImpliedDoListWhat : (*varRef* | *dataImpliedDo*) ;
varRef : (NAME | *arrayref*) ;
arrayref : #(NAME (*expr*)+) ;
externalFunction : #(EXTERNAL (*expr*)*) ;
intrinsicFunction : #(INTRINSIC (*expr*)*) ;
statementFunction : #(STFUNC (*expr*)*) ;
function : #(FUNCTION (*expr*)*) ;
expr : *operatorExpression* | *indirectValue* | *stringConstant* | *booleanConstant* | *arithmeticConstant* ;

nAryAddition : *expr* (*nAryAddition* |) ;

nAryMultiplication : *expr* (*nAryMultiplication* |) ;

pieces : *expr expr* ;

operatorExpression : #(COLON *expr expr*) | #(CONCATOP *expr expr*) | #(NEQV *expr expr*) | #(EQV *expr expr*) | #(LOR *expr expr*) | #(LAND *expr expr*) | #(LNOT *expr*) | #(IMPLIES *expr expr*) | #(LT *expr expr*) | #(LE *expr expr*) | #(EQ *expr expr*) | #(NE *expr expr*) | #(GT *expr expr*) | #(GE *expr expr*) | #(PLUS *aexpr nAryAddition*) | #(MINUS *expr expr*) | #(STAR *aexpr nAryMultiplication*) | #(DIV *expr expr*) | #(PLUS *expr*) | #(MINUS *expr*) | STAR | #(POWER *expr expr*) | #(SUM *expr* #(EQ *expr* #(RANGE *expr expr*))) | #(PIECEWISE (*pieces* | *expr*)+) | #(GAMMA (*expr*)+) | #(EXP *expr*) ;

indirectValue : *varRef* | *externalFunction* | *intrinsicFunction* | *statementFunction* | *function* ;

constant : *booleanConstant* | *arithmeticConstant* | *stringConstant* ;

booleanConstant : *trueconst* | *falseconst* ;

integerConstant : ICON | *zcon* ;

arithmeticConstant : *integerConstant* | RCON | *ccon* ;

stringConstant : *scon* | *hollerith* ;

hollerith : HOLLERITH ;

scon : SCON ;

ccon : #(CCON *expr expr*) ;

zcon : ZCON ;

trueconst : TRUE ;

falseconst : FALSE ;

A.3 List of Terminal Symbols

LABELREF	CONCATOP	THENBLOCK
XCON	CTRLDIRECT	ELSEIF
PCON	CTRLREC	ELSEBLOCK
FCON	TO	CODEROOT
RCON	SUBPROGRAMBLOCK	RECVAR
CCON	DOBLOCK	SUBPROGRAM
HOLLERITH	AIF	INTRINSIC

EXTERNAL
PARAMSUBPROGRAM
STFUNC
PIECEWISE
GAMMA
EXP
FACTORIAL
IMPLIES
SUM
RANGE
EQUALITY
FUNCTION
PARALLEL
VECTOR
DIMX
DIMY
PROJ
DOTPROD
EVAL
OVERWRITE
FORALL
COMMENT
"program"
NAME
EOS
"entry"
LPAREN
RPAREN
"function"
"block"
"subroutine"
COMMA
LABEL
"end"
"dimension"
"real"
COLON
STAR
"equivalence"
"common"
DIV
"complex"
ICON
"double"
"precision"
"integer"
"logical"
"pointer"
"implicit"
"none"
MINUS
"character"
"parameter"
ASSIGN
"external"
"intrinsic"
"save"
"data"
"assign"
"goto"
"go"
"if"
"then"
"elseif"
"else"
"endif"
"do"
"enddo"
"continue"
"stop"
"pause"
"write"
"read"
"print"
SCON
"open"
"fmt"
"unit"
"err"
"iostat"
"file"
"status"
"access"
"position"
"form"
"recl"
"blank"
"exist"
"opened"
"number"
"named"
"name"
"sequential"
"formatted"
"unformatted"
"nextrec"
"close"
"inquire"
"backspace"
"endfile"
"rewind"
"format"
DOLLAR
PLUS
"let"
"call"
"return"
NEQV
EQV
LOR
LAND
LNOT
LT
LE
EQ
NE
GT
GE
POWER
TRUE

FALSE	WHITE	NOTNL
XOR	ALPHA	INTVAL
EOR	NUM	FDESC
CONTINUATION	ALNUM	EXPON
WS	HEX	
ZCON	SIGN	

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide.* Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[3] A. N. S. I. (ANSI), *FORTRAN 77 Full Language (ANSI X3J3/90.4).* World Wide Web, 1978. `http://www.fortran.com/fortran/F77_std/rjcnf0001.html`.

[4] B. S. Baker, "An algorithm for structuring flowgraphs," *J. ACM*, vol. 24, no. 1, pp. 98–120, 1977.

[5] T. J. Biggerstaff, "Design recovery for maintenance and reuse," *Computer*, vol. 22, no. 7, pp. 36–49, 1989.

[6] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, (Los Alamitos, CA, USA), pp. 482–498, IEEE Computer Society Press, 1993.

[7] S. Blazy and P. Facon, "Partial evaluation for program comprehension," *ACM Comput. Surv.*, vol. 30, no. 3es, p. 17, 1998.

[8] E. Chikofsky and J. I. Cross, "Reverse engineering and design recovery: a taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, January 1990.

[9] P. Chowdhury, "Symbolic interpretation of legacy assembly language," Master's thesis, McMaster University, 2005.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* MIT Press, McGraw Hill, 2nd ed., 2001.

[11] D. E. Corporation, *VAX-11 FORTRAN Language Reference Manual.* Digital Equipment Corporation, 1982.

[12] G. Coschi and J. B. Schueler, *WATFOR-77 Language Reference Manual.* WATCOM Publications Limited, 1985.

[13] E. W. Dijkstra, "Go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, March 1968.

[14] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[15] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, no. 1, pp. 1–17, 1988.

[16] T. Fahringer and B. Scholz, *Advanced Symbolic Analysis for Compilers.* Springer-Verlag, 2003.

[17] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, "A fortran-to-c converter," tech. rep., Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, 1990.

[18] F. S. F. GNU Project, *Fortran.* `http://www.gnu.org/software/fortran/`: World Wide Web, December 1999. Last Viewed June 2006.

[19] G. H. Golub and C. F. V. Loan, *Matrix Computatations.* Baltimore, Maryland: John Hopkins University Press, 3rd ed., 1996.

[20] M. T. Harandi and J. Q. Ning, "Knowledge-based program analysis," *IEEE Softw.*, vol. 7, no. 1, pp. 74–81, 1990.

[21] M. S. Hecht and J. D. Ullman, "Flow graph reducibility," in *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 238–250, ACM Press, 1972.

[22] G. Inc., *CodeSurfer.* `http://www.grammatech.com/products/codesurfer/`: World Wide Web, Published Year N/A. Last Viewed March 2006.

[23] ISO/IEC, "Information technology – syntactic metalanguage – extended BNF," tech. rep., ISO/IEC, 1996. Reference Number: ISO/IEC 14977:1996.

[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[25] E. Moretti, G. Chanteperdrix, and A. Osorio, "New algorithms for control-flow graph structuring," in *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, (Washington, DC, USA), p. 184, IEEE Computer Society, 2001.

[26] Netlib, *BLAS (Basic Linear Algebra Subprograms).* `http://www.netlib.org/blas/`: World Wide Web, Published Year N/A. Last Viewed January 2006.

[27] Netlib, *LAPACK - Linear Algebra PACKage.* `http://www.netlib.org/lapack/`: World Wide Web, Published Year N/A. Last Viewed January 2006.

[28] T. J. Parr, "An overview of SORCERER – a simple tree-parser generator." Can be found online at `http://www.antlr.org/papers/sorcerer.ps`, 1994.

[29] T. J. Parr, *Translators Should Use Tree Grammars.* `http://www.antlr.org/article/1100569809276/use.tree.grammars.tml`: World Wide Web, November 15 1994. Last Viewed March 2006.

[30] T. J. Parr, *ANTLR, ANother Tool for Language Recognition.* `http://www.antlr.org/`: World Wide Web, Published Year N/A. Last Viewed December 2005.

[31] O. Signore and M. Loffredo, "Charon: a tool for code redocumentation and re-engineering," in *Proceedings., IEEE Second Workshop on Program Comprehension, 1993.*, pp. 169 – 176, July 1993.

[32] R. Tarjan, "Testing flow graph reducibility," in *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 96–107, ACM Press, 1973.

[33] W. Waite, *FORTRAN Syntactic Analysis Specification.* `http://eli-project.sourceforge.net/fortran_html/Parse.html`: World Wide Web, Published Year N/A. Last Viewed December 2005.

[34] Y. Wang, "Towards automated construction of tabular expressions," Master's thesis, McMaster University, 2006.

[35] M. Ward, F. Calliss, and M. Munro, "The maintainer's assistant," in *Conference on Software Maintenance*, pp. 307–315, October 1989.

[36] M. Ward, "Abstracting a Specification from Code," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 2, pp. 101–122, June 1993.

[37] M. Ward and H. Zedan, "MetaWSL and meta-transformations in the FermaT transformation system," in *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, (Washington, DC, USA), pp. 233–238, IEEE Computer Society, 2005.

[38] M. Ward, "Reverse engineering from assembler to formal specifications via program transformations," in *Seventh Working Conference on Reverse Engineering*, pp. 11–20, November 2000.

[39] A. Wassyng and M. Lawford, "Lessons learned from a successful implementation of formal methods in an industrial project," in *FME 2003: International Symposium of Formal Methods Europe* (K. Arakai, S. Gnesi, and D. Mandrioli, eds.), vol. LNCS Vol. 2805, (Pisa, Italy), pp. 133–153, Springer-Verlag, September 2003.

[40] Y. Zhai, "An analysis of program by symbolic computation," Master's thesis, McMaster University, 2006.