A Fortran Parsing Tool to Extract Parallelising Information

by

Miguel Nuno Fialho dos Santos Nicolau
BSc

Master of Science in Computer Systems
University of Limerick

Supervised by: Dr. Conor Ryan

Submitted to the University of Limerick, October, 2002
Abstract

This work describes the development of a Fortran parsing tool, which extracts parallelising information from legacy code. This tool will be used as a front-end parser for an evolutionary algorithm, which evolves sequences of parallel transformations to apply to serial code. As only data dependencies are considered in this process, the parsing tool is required to encapsulate flow dependencies in that information, in a transparent manner, to prevent the evolution of transformations that change the semantic meaning of the original serial code.

The techniques employed to handle this constraint, as well as other techniques required for the correct extraction of data dependencies, are described in this work, as well as the choices and practical details to consider when building a new parsing tool.
Acknowledgements

It is a pleasure to finally write some words of acknowledgment, to thank everyone who helped me in various ways to complete the work described in this thesis.

First and foremost, I would like to thank Dr. Conor Ryan, who always stood by my side during all these turbulent years, and who provided me not only with his knowledge and expertise, but also moral support and friendship.

Prof. Anthony Cahill was a source of knowledge I came to rely on. His expertise in the field opened my eyes to many of the problems ahead, and his ideas gave me hints on how to avoid those problems.

Dr. Michael O’Neill was always a friend I could count on while doing this work, and kindly proofread this document at short notice. His company and support from the start made me feel right at home.

There are some people in the Computer Science and Information Systems Department who I must thank for their support. Dr. Ita Richardson, head of Department, who believed in my work and made sure I was cared for in the later stages of my work. Dr. Jim Buckley and Michael English, who were two friends I could always rely on. The administrative staff of the C.S.I.S. department, namely Bianca Pluk, Gemma Ryan, Maeve Gleeson, Maria O’Connell and Nuala Kitson, must be thanked for their patience and support.

I must thank my Mother, for her endless support throughout my Irish years, which will no doubt continue for the years to come, regardless of where I will land tomorrow. Likewise to my Father, who has always provided
a helping hand in times of need. My brother João and his wife Filipa were always a source of support, and all the weekends spent in Dublin provided a haven to escape the pressure of work. Most especially, I must thank Andrea for love and support, and for enduring my irregular life style when writing this thesis.

Je dois aussi remercier mes copains belges, surtout Steve Servais, Quentin Lambeau, Rudolf Geets, Nicolas Senez et Aline Dero, qui m’ont toujours supporté depuis mon départ de Belgique.

This research was partially supported by the National Software Directorate.
Contents

1 Introduction .......................... 1
  1.1 Auto-parallelisation of legacy code .................. 1
  1.2 Research contributions .......................... 2
  1.3 Organisation .............................. 3

2 Background ........................... 5
  2.1 Software re-engineering .......................... 5
    2.1.1 Software maintenance ...................... 6
    2.1.2 The re-engineering process ............... 7
  2.2 Auto-parallelisation .......................... 7
    2.2.1 Hardware ................................ 7
    2.2.2 Parallel computing ....................... 8
    2.2.3 Automatic parallelisation ............... 11
  2.3 Automatic information extraction ............... 12
    2.3.1 Parallelising information ............... 12
    2.3.2 Parsing code ............................ 13
  2.4 Conclusion ................................ 15

3 S.C.A.R.E. ............................ 16
  3.1 Overview of the project ...................... 16
    3.1.1 SoftDraw ............................... 16
    3.1.2 Paragen ............................... 18
    3.1.3 SoftPlan .............................. 18
  3.2 Paragen .................................. 18
3.2.1 Paragen I .............................................. 19
3.2.2 Real world issues ................................. 20
3.2.3 Paragen II ........................................... 22
3.3 The need for a pre-processor ...................... 26
3.4 Conclusion ............................................. 27

4 Specification ............................................ 29
4.1 Integrating the SCARE project ..................... 29
  4.1.1 Language specific pre-processors ................. 30
  4.1.2 Choosing a language ................................ 31
4.2 Fortran .................................................. 32
  4.2.1 Brief history of Fortran ............................ 32
  4.2.2 The choice of FORTRAN 77 ................. 35
  4.2.3 Fortran analysis overview ....................... 35
4.3 Interface specification ............................... 37
  4.3.1 Input .............................................. 37
  4.3.2 Output ............................................ 38
  4.3.3 Example ........................................... 39
4.4 Conclusion ............................................. 41

5 Previous work ........................................... 42
5.1 Existing tools ........................................ 42
  5.1.1 Parafrase-2 ........................................ 42
  5.1.2 Polaris ............................................ 43
  5.1.3 ParaScope ......................................... 45
  5.1.4 SUIF ............................................... 46
  5.1.5 VFCS .............................................. 47
5.2 Why develop an entirely new tool .................. 48
5.3 Conclusion ............................................. 49

6 Practical issues ....................................... 50
6.1 Program unit inter-communication .................. 51
List of Figures

2.1 Flynn’s taxonomy classification ........................................ 10
2.2 A simplistic view of an information extraction tool ............ 14

3.1 Overview of the SCARE project ........................................ 17
3.2 Critical paths in a program ............................................. 20
3.3 Regression testing of a program ...................................... 21
3.4 Sequencing transformations in Paragen II ........................ 22
3.5 An evolved sequence of atom transformations in Paragen II 25
3.6 How Loop Shrinking works ............................................ 26

4.1 Language specific pre-processors ..................................... 30
4.2 The evolution of the Fortran language ............................... 34

5.1 Overview of the Parafrase-2 compiler ............................... 43
5.2 Overview of the Polaris compiler ................................... 44
5.3 Overview of the ParaScope environment ............................ 45
5.4 Overview of the SUIF compiler framework ........................ 46

6.1 Classification of different types of procedures in FORTRAN 77 51
6.2 Example call graph for a FORTRAN program ...................... 53
6.3 Mapping entities in memory with equivalence statements ...... 78
6.4 Mapping array elements in memory with equivalence statements 79

7.1 Parse tree for an example sentence ................................... 92
# List of Tables

2.1 Example instructions with *DEF* and *USE* sets ........................... 12

4.1 Example FORTRAN 77 program ........................................... 39
4.2 Parallelising information from a FORTRAN 77 source code .... 40
4.3 Specific loop information ....................................................... 41

6.1 Order of statements in FORTRAN 77 standard ......................... 56
6.2 Applying priorities to the ordering of statements ................. 58
6.3 *DEF* and *USE* sets with labelling technique ......................... 59
6.4 Example program using unconditional *GO TO* statements .... 62
6.5 Semantic changes due to use of *GO TO* statements ............... 63
6.6 *DEF* and *USE* sets with unconditional *GO TO* statements ... 64
6.7 Mapping names to entities in common blocks ......................... 75
6.8 Result of calling different entry points in a program unit .... 83

7.1 Example test file from the Papa test suite ......................... 104
7.2 Output of Papa for the test file ........................................ 105
7.3 Specific loop information for the test file ......................... 105
Chapter 1

Introduction

This thesis describes the analysis and development of Papa (Paragen Pre-processor and Analyser), a tool which accepts a program written in a legacy programming language, and extracts relevant information regarding the auto-parallelisation of that program. It is part of the Soft Computing Applied to Re-Engineering (S.C.A.R.E.) project, which aims to automatically transform a legacy serial program into a semantically equivalent parallel program.

1.1 Auto-parallelisation of legacy code

The last 40 years have seen a revolution in the role of software in our society. Many companies nowadays completely depend on their software systems, which elevates the role of software maintenance to a vital one. Some of those systems are known as legacy systems, i.e. systems that were developed a long time ago, and require constant updates, from changing customer needs to porting those systems to new hardware.

In recent years, software re-engineering has emerged as a field which deals with examining existing applications, and applying any required changes. This is, however, a tedious and strenuous task, as analysts often have to examine code that was written many years ago, when issues such as structured programming and documentation were not considered important.
It comes therefore as no surprise that one of the more promising fields in software re-engineering is that of the automatic parallelisation of code. With the recent availability of affordable parallel processing hardware, automated software re-engineering tools accept serial legacy code as input, and create a functionally equivalent parallel program as output, which can take advantage of the new hardware.

The SCARE project is one such project. Using diverse technology such as compiler theory, evolutionary computation, and message passing techniques, it provides a full framework for transforming existing legacy code onto a new parallel program, which maintains the semantic meaning of the original serial code.

The tool **Papa**, described in this thesis, is used at the initial stages of that process. Its main goal is to accept the original serial code as input, and extract parallelising information that is needed by other components of the project. This works involves a deep knowledge of the programming language used on the code, to ensure the validity of the output information.

## 1.2 Research contributions

During the analysis that lead to the development of **Papa**, and during its implementation, many problems and obstacles were found, which are likely to confront other projects involving the development of similar tools; the solutions employed, described in this thesis, present generalised or specific solutions to those problems, and are likely to be of use to other similar projects.

The following is a list of the major contributions of the work presented:

- **Papa** is an essential component of the SCARE project, and as such its development greatly contributed to the completion of that project. It is a fully developed tool, ready to use, and is the first language-specific pre-processor available for the project.
• The output provided by Papa is tied to a specific format required by other SCARE tools; as that format is composed of language-independent data dependencies, Papa must encapsulate all other information on that format. The techniques used for the encapsulation of language-specific information, introduced in this thesis, can be of use when extracting the same information from other programming languages, thus lightening the burden of developing similar tools for other programming languages, to work in the SCARE project.

• Although specifically developed to integrate the SCARE project, the work presented in this thesis will be of use to most software re-engineering projects working with legacy Fortran code, as it provides an in-depth analysis of the language, concerning the extraction of parallelising information.

1.3 Organisation

The following chapter describes the background and the motivation for this work, giving a brief introduction to the fields of Software Re-Engineering, Parallel Computing, and Auto-Parallelisation, before concentrating on the automatic extraction of parallelising information.

Chapter 3 gives an overview of the SCARE project, describing its components, particularly those that are directly related to Papa, and highlights the need for the work presented in this thesis.

In Chapter 4, the choice of designing the pre-processor for FORTRAN 77 is justified, and the interface of the tool is specified, that is, its input and output are clearly described; an example of the information to extract from a sample original serial program is also provided.

Chapter 5 looks at some of the existing tools that perform a similar task to that of Papa, and justifies the decision to develop an entirely new tool.

The practical issues concerning the implementation of this work are described in Chapter 6, which looks in detail at most of the FORTRAN 77
constructs, specifically those that require extra analysis concerning the extraction of data dependencies from the code.

In Chapter 7, the development choices for Papa are justified, and the techniques employed when extracting the required information from the code are described.

Finally, Chapter 8 summarises the work presented, and suggests future directions for research and improvement.
Chapter 2

Background

In this chapter, the context and consequently the need for this project are analysed. Rather than being a detailed description of the fields of software re-engineering, auto-parallelisation and automatic information extraction, this chapter aims at giving a brief introduction to these subjects, highlighting the relevant topics to this project.

2.1 Software re-engineering

As software systems increasingly become the core of a company’s activity, there is a pressing demand for legacy systems to be at all times current and operational. This usually involves maintaining and updating systems that were written at a time when the main qualities of a program were its speed and memory requirements, and issues such as structured programming and documentation were not of major importance (Bennett, 1991). The massive problems with maintenance and enhancement brought some enterprises, as early as 1993, to the level where their entire workload is related to updating, enhancing and fixing problems in existing legacy applications (Jones, 1994).
2.1.1 Software maintenance

Such is the importance of software for today’s business, that the maintenance and upkeep of that software has become critical to the survival of most companies. Software maintenance therefore plays a very important role on the success of a company. With the aging of that software, its maintenance becomes more and more expensive: it is estimated that up to 60% or more of a software engineer’s time is spent on information searching for program understanding and maintenance tasks (Pressman, 2000).

In a typical development cycle for a software system, a product is said to be complete when it is delivered to the customer or client, and the software is installed and released for operational use. All activities undertaken on the software thereafter are said to be software maintenance tasks. Software maintenance usually takes three main forms (Lientz & Swanson, 1980):

Corrective maintenance is the process of diagnosing and correcting errors in the software; these corrections are generally known as bug-fixes, but they could also be due to incomplete or inconsistent requirements.

Perfective maintenance is required when a program is successful, but as the system is used, new needs and requirements are discovered, and pressure is brought upon the developers to update and extend the functionality of the system.

Adaptive maintenance is needed when the environment in which the system operates changes; this could be a small change in the operating system, such as the introduction of a new version or the removal of support for existing facilities, or a bigger change, such as the migration to a new environment, or to new hardware.

A fourth form of maintenance, known as Preventative maintenance (Martin & McClure, 1983) also exists, which can be seen as a radical form of perfective maintenance; it takes a legacy system and converts its structure, or translates it to a new language, using the old system as the specification for the new system. It is usually referred to as Software re-engineering.
2.1.2 The re-engineering process

Software re-engineering is looked upon as a solution to many of the problems of maintaining existing software systems. Many different software re-engineering definitions exist, and the definition one uses depends on the perspective from which the re-engineering activity is looked upon. The definition used in this document is that software re-engineering is

“the process of modifying the internal mechanisms of a system or program, or the data structures of a system or program, without changing its functionality.” (GUIDE, 1989)

A distinction exists between re-engineering and reverse engineering, although these two terms are sometimes confused. The re-engineering process changes the underlying technology of a system, without affecting its main functionality; reverse engineering on the other hand is the backward engineering of a system to the specification stage, allowing the system to be regenerated more economically and productively (Yourdon, 1989).

One of the most exciting research fields in software re-engineering is the automatic parallelisation of code. Fuelled by the enormous advances that have been made in parallel architectures, particularly in the past decade, the re-engineering of legacy code to take advantage of these architectures has become a field in constant motion, and a source of both research and commercial success (Polychronopoulos et al., 1989; Blume et al., 1996; Cooper et al., 1993; Wilson et al., 1994; Chapman et al., 1992).

2.2 Auto-parallelisation

2.2.1 Hardware

In the early days of computer manufacturing, even before transistors replaced vacuum tubes, the main design adhered to was that of a single processor executing a single instruction at a time. Sometimes referred to as the John
van Neumann model, it is a simple design that is economically viable, and is one of the main reasons for the spread of computers as accessible tools.

Even in those early ages, the advantages of parallel computers were obvious; however, the complexity of designing and programming such machines, along with the cost of their development, has meant that, until recently, that technology has not been able to fulfil its promises. Also, the continuous advances in electronics, with cheaper, faster and more reliable components available at an astonishing rate, has provided enough power to single processor computers, at a fraction of the cost.\(^1\)

The availability of such hardware has, however, recently refuelled the field of parallel computation, with more and more multi-processor computers becoming publicly available. In particular, Beowulf clusters, through the use of technologies such as PVM (Geist et al., 1994) and MPI (Snir et al., 1995), have made the advantages of parallel computing available at very competitive rates, providing a flexible architecture. On the other hand, there is a physical limit to the speed a silicon based micro-chip can achieve, whereas theoretically there is no bound to the speed-up achieved by simply adding more nodes to a cluster of parallel running machines.

Ideally, a multi-processor computer with \(n\) processors should increase the speed of execution of a program by \(n\) times. In practice, however, the complexity of resource sharing, synchronisation and scheduling, scales down the processing power, and highlights the difficulties of parallel computing.

### 2.2.2 Parallel computing

A number of different architectures for the construction of parallel computers have been proposed. The most widely used classification for such architectures is the one proposed by Michael Flynn (Flynn, 1972), who has given a classification based on the way data and instruction streams are processed. Flynn defines four main categories (illustrated in Figure 2.1):

\(^1\)It is estimated that, in the late 1990s, computer performance was doubling every 12 months (Moravec, 1998).
SISD (single instruction stream, single data stream) is the traditional single sequence machine, based on the von Neumann uni-processor model, where a single stream of instructions act upon a single stream of data; an algorithm running on a computer of this class is known as a sequential algorithm, as the instructions are executed one at a time, in a sequential fashion.

SIMD (single instruction stream, multiple data streams) is a model where a single sequence of instructions is applied to multiple independent data streams; computers with this architecture will have each processor executing the same instruction at the same time, but each on its own data. Computers based on this model are usually referred to as a vector machines.

MISD (multiple instruction streams, single data stream) describes an architecture in which a machine executes different instructions on the same data moving as a stream; in this model, parallelism comes from having each processor performing a different operation on the same data, placed on a shared common memory.

MIMD (multiple instruction streams, multiple data streams) is a model that describes machines that can execute different types of instructions on different sets of data; each processor executes a different instruction on different data. Memory can be shared, through the use of a global memory accessible by all processors, or distributed, where each processor has its own local memory, and processors communicate through message passing.

In addition to Flynn’s classification, a fifth class known as SPMD (single program, multiple data) describes a special case of SIMD/MIMD, in which all processors execute the same code on different data sets, although at any given time the instruction executed by each processor can be different (Almasi & Gottlieb, 1994).
Figure 2.1: Flynn’s taxonomy classification, showing all combinations of single or multiple instruction and data streams, between control units and processing elements.

From the models described, SIMD and MIMD are the most widely used in parallel computing, whereas MISD is the least popular (Akl, 1989). The SIMD model is well suited for specific tasks, such as data parallel algorithms, where the operations to be performed are uniform across the data set (the algorithms are said to exhibit a high degree of uniformity (Zomaya, 1996)); the advantages that this model presents include implicit synchronisation, and the need for only one program.
MIMD architectures have the advantage of being far more flexible, in that they can also be applied to the same tasks as the SIMD model, but their multiple streams of instructions allow multiple threads of control, enabling a much broader application range. Computers based on this model are naturally harder to program, due to their architectural complexity; indeed, parallel programming requires in-depth knowledge of the underlying multiprocessor system, and is difficult and error prone. Furthermore, the variety of existing architectures often makes it more difficult to port a program than to reprogram it completely.

These problems suggest that it would be desirable to let programmers develop their systems using sequential code, which better matches the human way of reasoning, and to have an automatic process of parallelisation, dependent on the architecture used. The existence of such a process would also allow the re-engineering of legacy code, written in a serial manner, to be automatically parallelised, so as to take advantage of the hardware now available.

2.2.3 Automatic parallelisation

An automatic parallelising compiler is a tool that searches for potential parallelism in a sequential program, and generates parallelised code for a given multi-processor system. Common techniques in compiler code optimisation for parallel architectures include explicit and implicit parallelisation: explicit parallelisation is achieved through the use of a programming language’s built-in directives, that allow the programmer to mark sections of the code as possible to parallelise; implicit parallelisation is achieved without human intervention, through an extensive analysis of the serial code to parallelise.

Since there is a vast amount of legacy code already written in a serial fashion, the process to rewrite it so that it runs in parallel architectures should be an automatic one, as part of the re-engineering process. This means that parallelising information should be automatically extracted from the source code, to enable the implicit parallelisation process.
2.3 Automatic information extraction

2.3.1 Parallelising information

A normal sequential program is composed of a list of instructions, which are to be executed one after the other. In order to parallelise such a code, those instructions need to be regrouped, such that the instructions within a group can be executed at the same time, two or more groups of instructions can be executed independently of each other, two or more groups can be overlapped, or any combination of these (Banerjee et al., 1993). In fact, the instructions of a serial program can be executed in any order, as long as the meaning of the original program remains intact.

In order to execute instructions in parallel, the data dependencies between those instructions need to be discovered and analysed. If there is no chain of dependence between two instructions, that means they can be executed in parallel (Bacon et al., 1994).

Data dependencies are derived from the calculation of DEF and USE sets of each instruction in a serial program. Given a statement $S$, its DEF set is the set of variables written to (i.e. DEFined) with the execution of $S$; its USE set is the set of variables USEd in the execution of $S$. Table 2.1 shows some examples, including a case when a variable is a member of both the DEF and USE sets.

Table 2.1: Example instructions with DEF and USE sets.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>DEF</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = Y</td>
<td>{X}</td>
<td>{Y}</td>
</tr>
<tr>
<td>X = 3</td>
<td>{X}</td>
<td>{}</td>
</tr>
<tr>
<td>X = X * Y</td>
<td>{X}</td>
<td>{X, Y}</td>
</tr>
<tr>
<td>DO 200 I=1,N</td>
<td>{I}</td>
<td>{N}</td>
</tr>
<tr>
<td>IF (A(I) .GT. 3) THEN</td>
<td>{}</td>
<td>{A, I}</td>
</tr>
</tbody>
</table>
If the order of execution of two statements will not affect the final output of a program, then there are no data dependencies between those two statements. The types of dependencies that can occur between statements are:

**Flow dependency** : occurs when a variable $X$ exists in the $DEF$ set of a statement $S_1$ and in the $USE$ set of a later statement $S_2$, with no intervening redefinition of $X$; this means there is a flow dependency from $S_1$ to $S_2$.

**Anti dependency** : occurs when a variable $X$ exists in the $USE$ set of a statement $S_1$ and in the $DEF$ set of a later statement $S_2$, with no intervening redefinition of $X$; this means there is an anti dependency from $S_1$ to $S_2$.

**Input dependency** : occurs when a variable $X$ exists in the $USE$ set of a statement $S_1$ and in the $USE$ set of a later statement $S_2$, with no intervening redefinition of $X$; in this case, there is an input dependency from $S_1$ to $S_2$.

**Output dependency** : occurs when a variable $X$ exists in the $DEF$ set of a statement $S_1$ and in the $DEF$ set of a later statement $S_2$, with no intervening redefinition of $X$; in this case, there is an output dependency from $S_1$ to $S_2$.

In order to extract these dependencies from serial code, an analysis of the used language is required. This is an easily automated process, through the use of a parser for each specific language used in the legacy code.

### 2.3.2 Parsing code

Ever since the programming of computers became more and more accessible, new programming paradigms have seen the light of day. Examples of programming paradigms include structured programming, or, more recently,
object-oriented programming. This fact, along with the appearance of new technologies, and also the great diversity of ability and personal taste of programmers, has led to the existence of several hundred programming languages nowadays (Grassman & Tremblay, 1996; Naiditch, 1999).

Grammars have been extensively used to describe and design programming languages. A grammar defines a language through the use of a finite set of grammatical rules, that give structure to the sentences of a language. By mechanically analysing and matching the rules described in the grammar, a program can be syntactically validated (i.e. its syntax can be verified).

Compilers make use of such grammars to parse source code. A compiler can be described as being

“a program which translates a set of instructions (a program) in one language into some other representation; typically, the output of a compiler is in the native binary language that can be run directly on a computer.” (Levine et al., 1995)

Therefore, the extraction of parallelising information from a source program can be seen as a compiling process. Figure 2.2 shows a basic structure of such a process; it is comprised of the following three steps:

**Lexical analysis**: is performed by a scanner, whose purpose is to separate the incoming text into tokens, such as constants, variables, keywords and operators; depending on the language, this type of analysis is usually quite simple to perform.
**Syntax analysis**: is the process whereby the sequence of the tokens provided by the scanner is matched against the grammar for the given language; this is usually a complex but needed process, to discover the meaning of the original tokens (e.g. *defined* versus *used* variables).

**Code generation**: is the process through which the syntax tree and symbol table (amongst other data provided by the syntax analysis) are used to generate the required information (in this case, the information needed to parallelise the original code); it is sometimes called the transformation phase, if parallel transformations are applied to the original source code.

### 2.4 Conclusion

This chapter has taken us on a journey through the needs of software maintenance and the promises of software re-engineering. The benefits of the parallelisation of legacy code, exploiting the emergence of the available hardware, have been shown, and the need for the automation of this process has been highlighted.

Finally, an introduction to the automatic extraction of parallelising information from legacy code has been given, along with an overview of the steps usually taken in this process.
Chapter 3

S.C.A.R.E.

In 1997, the Soft Computing And Re-Engineering (SCARE) group, of the University of Limerick, started working on a project concerned with the development and integration of the concepts for serial to parallel code transformation (Soft Computing and Re-Engineering Group, 1997). This chapter gives an overview of that project, concentrating on the components that are directly related to Papa, and describes the need for Papa in the context of the project.

3.1 Overview of the project

The problem tackled by the SCARE project is that of creating a framework for converting serial programs into functionally identical parallel programs. The project is composed of three parallel streams, illustrated in Figure 3.1, and these are briefly described in this section.

3.1.1 SoftDraw

The first stream, SoftDraw, concentrates on the task of extracting information about the underlying logic and structure of the program about to be converted. This involves drawing graphs that model the original program,
Figure 3.1: Overview of the SCARE project: automatically transforming a serial program into a semantically equivalent parallel program.

and the relationships between its procedures, in the form of nodes, edges, and node-edge relationships.

This stream bases itself on graph drawing and software comprehension theories, to establish software representations, as well as other diagrams and semantic representations; these include call graphs for the procedures of the original program, and profile tables for each procedure existing in the original code.¹

Based on the information collected, SoftDraw can then make a decision about which procedures should be parallelised, and uses Paragen to parallelise those procedures.

¹A profile table is a table which lists an estimate of how many CPU cycles are needed to execute an instruction (or a whole procedure).
3.1.2 Paragen

The second stream, Paragen, is concerned with the re-engineering of the serial code, using the input provided by SoftDraw. The resulting parallel code must preserve the semantic meaning of the original serial code, and will be tested using the third stream, SoftPlan. Since Papa is a preprocessor for this stream, Paragen is described in more detail in Section 3.2.

3.1.3 SoftPlan

To test the efficiency of the serial to parallel transformation, and to run the parallel code, a virtual machine is required, and can be achieved by software such as PVM or MPI, as previously mentioned in Section 2.2.1. In order to benchmark and competitively test the parallel code, a scheduler is required, to efficiently allocate the computing resources available on the parallel machine to the parallel code segments. This third stream, Soft Plan, involves the development of a static scheduler, that will combine the explicit parallelism produced by Paragen and the resources available, with the objective of minimising the amount of time required to execute the parallel program; its output will be therefore a load balanced schedule.

3.2 Paragen

The Paragen stream (Ryan, 1996; Ryan, 1999) is focused on the conversion of the original serial code into a semantically equivalent parallel program, using the information supplied by the SoftDraw component.

Paragen uses Genetic Programming (Koza, 1992) to search the space of semantically equivalent programs, and choose one to use as a parallel version of the original serial code. Genetic Programming (GP) is useful in this context, due to its proven reliability across a wide range of problems. Using a population (collection) of individuals (programs), a subset of the set of all programs, GP evolves those individuals using the Darwinian principle
of survival of the fittest: using genetic operations such as crossover and mutation, individuals are combined and transformed, and are evaluated with a given fitness measure.

The approach taken with Paragen is that a program unit (i.e. the main program or a procedure) should stand on its own, and therefore it works by parallelising a single unit at a time. The way GP is used to evolve parallel versions of the original serial program is explained in the next three sections.

3.2.1 Paragen I

In its initial implementation (Walsh & Ryan, 1995; Walsh & Ryan, 1996), Paragen worked by disassembling each function into its individual instructions, and then by using GP to reassemble those original instructions in a parallel form (when possible), with the aim of obtaining a faster, but semantically correct, program.

To evaluate a program regarding the speed of its execution, the duration of the critical path through the program was calculated. The critical path of a program is the longest possible path of instructions that must be executed in a function. To calculate the critical path, the execution times of each instruction, as well as the communication times between them, must be added up. In the example given in Figure 3.2, the critical path is \(<I_1, I_2, I_4>\), because it is the longest of all possible paths.

The generated parallel program should not only be fast, but also semantically equivalent to the original program. To test for equivalence, a set of test cases was generated using the original program; specifically, on each test case, all the variables from the original function were given a random value, and the function was then executed. This generated a set of inputs with their corresponding correct outputs, which could then be applied to the generated parallel programs, as shown in Figure 3.3. Each generated parallel program could therefore be given a score for correctness, which was the number of correct variable values when given the test inputs; this kind of testing is known as “regression testing” (Rothermel & Harrold, 1996).
Figure 3.2: Calculating the critical path in a program; nodes represent instruction execution times, and edge values represent communication times.

Possible paths:

\[ I_1 \rightarrow I_2 \rightarrow I_4 \] (21 time steps)

\[ I_1 \rightarrow I_3 \rightarrow I_4 \] (19 time steps)

These two measures of fitness could then be combined, to give a score to each generated program; at the end of each GP run, these scores were used to evaluate all generated programs, and to select which ones would be used to create a new population of programs.

### 3.2.2 Real world issues

Paragen’s initial results were very encouraging, and the success enjoyed with all the experiments conducted suggested that it could be used as an alternative to traditional parallelisation methods. However, when considering its application to “real world” problems, the original implementation of Paragen ran into certain stumbling blocks. Two of the most important are the provability of the generated code, and the scalability of the system.

#### Provability

The technique to test the correctness of an evolved program, described in the previous section, never failed once in determining whether that program kept the semantic meaning of the original serial program. It is, however, based
Figure 3.3: Testing the correctness of a program; for the given set of inputs, a list of variable values is obtained as output, which can then be compared to the correct outputs.

on a set of test cases, and although this is generally proof enough for a wide range of applications, there are certain cases, such as air traffic controller software, or similar mission critical software, where that proof is simply not exhaustive enough.

The best way to prove the correctness of a program is by showing the sequence of transformations applied to the original serial code; the original implementation of Paragen was however incapable of doing this, as the code was broken down into a set of instructions, which were then rejoined together, through an evolution process that is impossible to track.

**Scalability**

The other issue is that of scalability. When Paragen evolved a new program, it had to execute it, to determine whether that program produced the correct set of variable values, given a set of test cases. When using small programs, this was not a problem, but generally, legacy software is often composed of thousands of lines of code, which would make testing each evolved program prohibitively time consuming.
Figure 3.4: Applying a sequence of transformations to a program using Paragen II.

\[ P' \rightarrow P'' \]

3.2.3 Paragen II

To deal with the issues raised in the previous section, Paragen II was developed (Ryan & Walsh, 1997). The approach taken was that, rather than evolving parallel versions of the original code, which require testing and benchmarking, Paragen would now evolve sequences of parallel transformations, which when applied to the serial code, will produce a parallel version of that code. Because these transformations are semantic preserving, the need for testing is removed; furthermore, because the sequence of transformations is known, the new code can be proven to be semantically equivalent to the original one.

The transformations employed are standard parallel transformations, and preserve the semantic of the underlying program, as long as there are no data dependencies in the program segment being modified. Starting with the original serial code, each transformation is applied in turn, creating one or more new program segments, and these new program segments are used as input for the next transformation, as shown in Figure 3.4.
Loops are treated differently than single instructions, not only because of their complexity, but also because they usually present the best opportunities for parallelisation. Therefore, the transformations to be applied depend on whether the target of the transformation is a group of one or more single instructions (referred to as atoms), or a group of one or more loops; this results in Paragen II using two modes when evolving a sequence of transformations: Atom Mode and Loop Mode.

**Atom Mode transformations**

There are four different classes of transformations for atoms defined in Paragen II:

**Pxx/Sxx** These transformations break the current program segment into two new segments; the percentage of instructions from the original segment that will be included in each new segment is defined by the \( xx \) part. For example, \( P66 \) will create two new segments: the first will contain two thirds of the original segment, whereas the second will contain the remaining third. The difference between the \( Pxx \) and \( Sxx \) transformations is that the first one will create two new segments that will be run in parallel, whereas the second one will create two segments that must be run in sequence.

**FPAR/LPAR/FSEQ/LSEQ** These transformations are extreme cases of the \( Pxx/Sxx \) class; \( Fxxx \) transformations will isolate the first instruction from the rest of the program sequence, whereas \( Lxxx \) will isolate the last instruction. \( PAR \) and \( SEQ \) determine if the isolated instruction should be executed in parallel or in sequence with the remaining instructions.

**SHIFT** This is a simple transformation that delays the execution of the current program segment by one time step; although not a parallelising transformation in itself, it can be useful to eliminate data dependencies (by pushing the execution of certain instructions forward), and therefore allow for the application of other transformations.
NULL/PARNULL  This final class of transformations is used to terminate the execution of code; NULL causes all remaining instructions in the current program segment to be executed in sequence, whereas PARNULL causes them to be executed in parallel.

An example of the application of a sequence of transformations evolved by Paragen II is shown in Figure 3.5. Starting with the original serial function (the sequence of instructions \(< I_1, I_2, I_3, I_4, I_5 >\), the FSEQ transformation is applied, causing the first instruction, \(I_1\), to be scheduled for execution before the remaining four instructions; these are then passed as a new segment to the next transformation, FPAR. This transformation causes the first instruction of the segment, \(I_2\), to be executed in parallel with the segment \(< I_3, I_4, I_5 >\). This segment is then subjected to the S50 transformation, which divides it roughly in half, creating two new segments to be executed in sequence, \(< I_3, I_4 >\) and \(< I_5 >\). The first segment is then subjected to the P50 transformation, which creates two new segments, \(< I_3 >\) and \(< I_4 >\), which will be executed in parallel. These are then terminated by a NULL transformation, and the same happens to the \(< I_5 >\) segment.

Loop Mode transformations

Since loops present a special case for parallelising transformations, Paragen II treats them separately (Ryan & Ivan, 1998). When an atom is encountered, Paragen determines first if that atom is a meta-loop (that is, a loop, a nested loop, or a sequence of loops); if so, then a special loop mode is entered, and loop transformations are applied to that meta-loop. Once finished, Paragen II returns to atom mode, and keeps applying transformations as before.

When loop mode is entered, a sequence of one or more transformations is applied to that meta-loop. Other than simply transforming a loop onto its parallel version (e.g. transforming a DO loop into a PARALLEL DO loop), all manner of modifications and alterations exist, which can be applied to meta-loops. A number of these has been detected and used as transformations in Paragen II; the following are a sample of those loop transformations:
Loop Fusion This is a loop specific transformation that merges two loops into a single loop (Lewis & El-Rewini, 1992); it requires both loops to be already parallelised, and to have the same number of iterations. If either or both of the loops are not already parallelised, Paragen will try to parallelise them first. If the loops do not have the same number of iterations, Paragen will try to assemble a parallel loop construction that will execute as much of both loops together as possible. This results in reduced overhead computation, and provides the possibility of parallelising the contents of both loops.

Loop Shrinking When a meta-loop has got cross iteration data dependencies, the Loop Shrinking transformation can be applied. It transforms the original loop into two loops: a normal serial loop, and a parallel loop inside it, eliminating the cross-iteration dependencies, as shown in Figure 3.6.
3.3 The need for a pre-processor

With both the Paragen I and Paragen II approaches, the detection of data dependencies is of the utmost importance. Taking the case of the latter, when transformations are applied, if certain data dependencies are broken, the evolved sequence of transformations is creating an illegal parallel program, and cannot be used as the final solution.

Detecting data dependencies is not a trivial task; for example, if a P50 transformation is applied to a program segment of \( n \) lines, then that transformation is legally applied only if:

- There are no flow dependencies between instruction \( i \) and instructions \( (n/2 + n \% 2 + 1) \) to \( (n/2 + n \% 2 + i) \);
- There are no anti dependencies between instruction \( i \) and instructions \( (n/2 + n \% 2 + 1) \) to \( (n/2 + n \% 2 + i) \).

The identification of \( DEF \) and \( USE \) sets, as seen in Section 2.3.1, is therefore needed for each line of the current program segment, to determine the existence or not of data dependencies that might compromise the semantic meaning of the original code, when transformations are applied.
Similarly, when loop mode is entered, certain information regarding the meta-loop being transformed is needed. For example, for a Loop Fusion transformation, the number of iterations of each loop, as well as its increment factor, are necessary.

This information should therefore be made available to Paragen, through a pre-processor that will use the original serial code as input. This pre-processor must make all the parallelising information available to Paragen, before sequences of transformations are tested and evolved.

The tool presented in this thesis, Papa, has the function of extracting the parallelising information needed by Paragen, for the application of transformations. There exists an early study by Joseph Gaffey (Gaffey, 1997) for the development of a front-end to Paragen, to extract sets of used and modified variables from C statements; the tool developed had however several limitations, namely:

- it did not provide information regarding nesting levels;
- it did not provide instruction specific information;
- it did not analyse loops.

Furthermore, the mentioned tool was a pre-processor specifically designed for extracting information from source code written using the C language, whereas Papa is a pre-processor for the Fortran language. The need for language specific pre-processors is explained in the next chapter.

### 3.4 Conclusion

In this chapter, the SCARE project has been presented. It is composed of three main streams, SoftDraw, Paragen, and SoftPlan, working together with the objective of automatically transforming a serial program into a semantically equivalent parallel program.
Two approaches have been taken for the transformation of a serial program onto a parallel program, Paragen I and Paragen II; the stumbling blocks of the original Paragen implementation have been described, and the new approach of evolving transformations, taken in Paragen II, has been explained, along with examples of its application.

Finally, the need for the project presented in this thesis has been highlighted, and its placement in the SCARE project has been made clear.
Chapter 4

Specification

Prior to developing Papa, certain choices and their associated issues had to be considered; for example, the choice of a programming language for the pre-processor to act upon, and the specific characteristics and difficulties that come with the analysis of code written in that language, along with the information flow through the program (i.e. its inputs and outputs). This chapter sets out the reasoning behind the choice of Fortran as the language of the original serial program, and the issues associated with that choice. It then proceeds in detailing the input accepted by Papa, and the exact information needed by Paragen, and thus the output of Papa.

4.1 Integrating the SCARE project

As Paragen needs parallelising information to be extracted from the source code, a tool that understands that code must be used. As programming languages can be very different in their syntax, so too is their analysis, and therefore different tools must be used for different languages. The need for these language specific pre-processors and the choice of a language for Papa are explained in the next pages.
4.1.1 Language specific pre-processors

One of the advantages of the SCARE project is that it is mostly language independent; indeed, the transformations applied by Paragen are not restricted to a specific programming language, but rather to a programming paradigm.

Paragen always needs the same information (which is described in Section 4.3.2) to evolve its sequences of transformations, regardless of the programming language being used. This raises the need for language specific pre-processors (as seen in Figure 4.1). These will have specific knowledge of the language’s constructs, and will be able to extract the information needed by Paragen to evolve its transformations.

A choice had to be made therefore about which language to write a pre-processor for. Given the objectives of the SCARE project, one should focus on programming languages that are frequently used on legacy code; the next section concentrates on the choice of one such language.
4.1.2 Choosing a language

The emergence of the first electronic computers in the early 1950s motivated the development of the first high level languages. These high-level programming languages (some of which are still in use today) started with MATHEMATIC and FORTRAN, developed in 1957 by Hopper and by Backus, respectively. The 1960s saw the appearance of languages such as ALGOL, APL, PL/1 and BASIC.

The emergence of integrated circuits around 1965 enabled large scale computing, which fuelled the appearance of structured programming, leading to the development in the late 1960s and 1970s of languages such as PASCAL and C, and later in the 1980s, Modula-2 and ADA.

The object-oriented paradigm caused the emergence of languages such as Smalltalk, C++ and Java, whereas the paradigm of logic programming led to the development of Prolog. Finally, functional programming developed with the introduction of LISP, and continued with SCHEME.

From this very brief introduction to programming languages, it can be seen that they have evolved over time to reflect the contemporary hardware availability and the evolution of programming paradigms. To choose a single programming language from such a variety of languages, the following properties must be fulfilled:

- it must be widely used in legacy code;
- it must be compatible with the type of transformations applied by Paragen.

From the languages briefly presented above, the Fortran language has been chosen, as it is one of the oldest programming languages still available today, and a multitude of legacy systems still use it (Snelting, 2000). The reasons for this choice, as well as the characteristics of the language, are discussed in the next section.
4.2 Fortran

The Fortran language has been in use ever since it was first introduced in 1957. Over the years, new functionalities have been added by the introduction of new standards, to keep the language alive and up-to-date with the current programming trends and needs, as well as with newer hardware (such as multi-processor and distributed architectures).

Fortran (FORmula TRANslation) is a scientific language. It has been used on many scientific applications over the years, such as kernel specifications, benchmark programs, medical software, and even space exploration (Fortran was used for the programming of some of the early planetary probes used by NASA (Page, 1995)), to name but a few.

4.2.1 Brief history of Fortran

The Fortran language was originally designed for the solution of problems involving numerical computation, and its development dates back to the 1950s, with the first Fortran system, FORTRAN I,\(^1\) being released in 1957, for the IBM 704.

In the early 1960s, as other vendors adapted the language for their own architectures, the need for a standard became evident. A standards committee was established in 1962, and the FORTRAN 66 standard was released four years later.

Unfortunately, this first standard did not gather the approval of all the manufacturers using the language, and this fuelled the need for a new standard; by 1978, the FORTRAN 77 standard was released (American National Standards Institute, 1978).

\(^1\)As original Fortran implementations used only capital letters, references to the language were done using the all-caps FORTRAN word. Nowadays, both forms of the name, FORTRAN and Fortran, are in use; the convention most commonly used (and adapted in this text) is that versions up to 1977 are referred to as FORTRAN, and later versions are referred to as Fortran 90, Fortran 95, etc.
Immediately after its introduction, the FORTRAN 77 standard was adopted and widely used. Many saw it, however, as an effort to create a standard as fast as possible, rather than an effort to evolve the language; as a result, many vendors started adding their own extensions to the language, urging the need for the introduction of a new standard.

However, due to a variety of reasons, including political trouble (Meek, 1990), the development of the new standard took much longer than it should, allowing other programming languages to evolve and compete with Fortran. For example, the system-programming language C, and its evolved variant C++, became more popular in the traditional strongholds of Fortran, the scientific and engineering worlds, in spite of it being non-computationally oriented.

Originally called Fortran 82, the new standard was never agreed by all parties involved; between featureists and generalists, and traditionalists and revisionists, no-one seemed to fully agree with the new proposals, and the search for a new standard went on during the 1980s, being referred to as Fortran 82, Fortran 8x, and Fortran 88.

Finally, in August 1991 the new Fortran language standard (known as Fortran 90) was published (ISO Publications, 1991). It was released with the intention of modernising the language, introducing new features to compete with other languages (such as pointers) or to exploit new hardware (such as explicit parallel instructions), but at the same time keeping a backwards compatibility to the FORTRAN 77 standard.

However, certain features of the FORTRAN 77 standard were marked as obsolete, and some of those were removed from the language when the Fortran 95 standard was released (ISO Publications, 1997). By this time, however, many extensions of Fortran had followed their own stream of development, and had been adopted by several constructors and research centers; example of such languages are Fortran-D (Fox et al., 1990; Hiranandani et al., 1992), HPF (High Performance Fortran Forum, 1992; Koelbel et al., 1993), or Vienna Fortran (Chapman et al., 1992).
Work is currently underway towards a new Fortran 200x standard (Fortran Standards Technical Committee, 2002). Fortran 200x will be a mostly upwardly-compatible extension of Fortran 95, adding, among other features, support for exception handling, object-oriented programming, and improved inter-operability with the C language.

A graph summarising the evolution of Fortran is show in Figure 4.2.
4.2.2 The choice of FORTRAN 77

Due to its remarkable longevity (it has been in use for over 45 years), the Fortran language was chosen for Papa, as there is an enormous amount of legacy software using it. From the maze of Fortran specifications and standards, however, a standard had to be chosen for the pre-processor described in this thesis. The FORTRAN 77 standard was chosen, because:

- it is the most accepted standard of Fortran;
- most legacy Fortran applications were written using this standard;
- it is backwards compatible with earlier Fortran versions;
- it is the standard that covers the widest range in the Fortran time-line.

The Fortran 90 standard was not chosen mainly because of its explicit parallel instructions, which are not compatible with Paragen’s implicit parallelism approach; in any case, major software written using the Fortran 90 standard did not appear until well after the standard was introduced in 1991, when it became widely accepted.

4.2.3 Fortran analysis overview

This section will concentrate on the analysis of FORTRAN 77 code, and in identifying the difficulties associated with it. For further information regarding the FORTRAN 77 standard, the reader is referred to one of the many textbooks and documents available (Balfour & Marwick, 1988; Silicone Graphics Inc., 1994; Chivers & Sleightholme, 1990; Page, 1995).

The analysis of code by a compiler is typically divided into three distinct phases (Aho et al., 1986):

1. lexical analysis to recognise the language’s tokens;
2. syntax analysis to recognise significant groupings of tokens;
3. semantic analysis to associate meaning to the resulting structure.
FORTRAN 77 was designed well before techniques were developed to separate and simplify each of these three stages. It contains certain aspects that nowadays would be considered awkward for code analysis:

- No reserved keywords;
- the assumption that each text line has exactly 72 characters;
- syntactical distinction between user and system defined keywords;
- spaces are meaningless outside character strings;
- implicit typing of variables;
- no nesting level defining character or keyword;
- position dependent meaning of all characters within the first columns;
- similar syntax for array references and function calls.

A typical example of the the way Fortran can be easily misinterpreted is taken from the failure of one of the early planetary probes launched by NASA (Page, 1995). The cause of the failure was eventually traced to a statement in its control software, similar to the following:

\[
\text{DO 15 I = 1.100}
\]

when what should have been written was

\[
\text{DO 15 I = 1,100}
\]

but somehow a dot had replaced the comma. Because Fortran ignores spaces, this was interpreted by the compiler as

\[
\text{DO15I=1.100}
\]
which is a perfectly valid definition and assignment statement for a variable called \texttt{DO15I}, rather than the \texttt{DO} loop initially intended.

On top of these syntax related problems, other more specific problems exist, dealing with the nature of certain FORTRAN 77 statements. The way all these problems were dealt with is detailed in Chapters 6 and 7.

### 4.3 Interface specification

As stated previously, the objective of Paragen is to evolve sequences of language independent transformations, that will be later applied to the original code. Therefore, \texttt{Papa} needs to provide Paragen with the information necessary to detect data dependencies between lines of code from the original code, so that Paragen can evolve transformations that preserve the semantic meaning of the that code.

This section concentrates therefore on the requirements of Paragen, and thus on the output required from \texttt{Papa}, and the input needed to generate that output.

#### 4.3.1 Input

As the standard chosen for Fortran was the FORTRAN 77 one, it is assumed that the input code will respect that standard. Nevertheless, much of the code written after the standard was first introduced used many of the extensions vendors included in their own compilers; these extensions were so often used that they became accepted by the majority of the compilers, even though they were not part of the original standard. The decision was made therefore to make use of a standard FORTRAN 77 grammar for the syntax analysis, but to extend it so that the most used extensions to the original standard are accepted in the code.

The extensions accepted by \texttt{Papa} are listed in Section 7.1.2.
4.3.2 Output

**Papa** performs a lexical and syntactic analysis to extract, for each line of the source code, the following information:

- line number on original source code;
- nesting level;\(^2\)
- number of elements in *DEF* set;
- number of elements in *USE* set;
- contents of *DEF* set;
- contents of *USE* set;
- available information regarding loops (i.e. iteration variable, its start value, its end value, and its increment value).

As SCARE operates on tested legacy code, it is assumed that the original serial source code is both syntactically and semantically correct. In other words, **Papa** will not attempt to detect and signal errors encountered during its lexical and syntax analysis; these can easily be detected by means of a compiler.

Nevertheless, as certain non-standard instructions might appear in the source code, the grammar used by **Papa** might not be able to correctly parse them, and will therefore signal that fact to Paragen by means of a pre-defined error flag, while continuing its processing job. The need for this signaling comes from the fact that, not knowing the syntax of a certain instruction, **Papa** will not be able to correctly detect if that instruction uses or modifies any variables, and could therefore be passing potentially incomplete or misleading information to Paragen.

\(^2\)The nesting level of a statement is the level in which a statement appears, in the hierarchical structure of a program (that is, its place inside the combined structure of *IF* statements, *DO* loops, etc).
4.3.3 Example

As an example of the information to be extracted with Papa, consider the example FORTRAN 77 code shown in Table 4.1, consisting of twenty lines of code. For each of those lines, Papa must extract the information described in Section 4.3.2, which for the example code is the one shown in Table 4.2. In Table 4.3, the specific information required for each loop is shown.

Table 4.1: Example FORTRAN 77 program.

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A(5)=A(4)+B(3)</td>
</tr>
<tr>
<td>2</td>
<td>COUNTER=COUNTER+1</td>
</tr>
<tr>
<td>3</td>
<td>INDEX=INDEX-1</td>
</tr>
<tr>
<td>4</td>
<td>DO 100 I=1,N</td>
</tr>
<tr>
<td>5</td>
<td>C(I)=X+Y+I</td>
</tr>
<tr>
<td>6</td>
<td>100 CONTINUE</td>
</tr>
<tr>
<td>7</td>
<td>DO 200 I=1,H</td>
</tr>
<tr>
<td>8</td>
<td>B(I)=M(I)+X+I</td>
</tr>
<tr>
<td>9</td>
<td>200 CONTINUE</td>
</tr>
<tr>
<td>10</td>
<td>DO 300 I=1,N+1</td>
</tr>
<tr>
<td>11</td>
<td>A(I)=X+A(I)+1</td>
</tr>
<tr>
<td>12</td>
<td>300 CONTINUE</td>
</tr>
<tr>
<td>13</td>
<td>B(3)=B(0)+A(0)</td>
</tr>
<tr>
<td>14</td>
<td>DO 400 I=2,N</td>
</tr>
<tr>
<td>15</td>
<td>DO 410 J=2,N</td>
</tr>
<tr>
<td>16</td>
<td>D(I,J)=(D(I+1,J)*D(I,J+1))/INDEX</td>
</tr>
<tr>
<td>17</td>
<td>410 CONTINUE</td>
</tr>
<tr>
<td>18</td>
<td>400 CONTINUE</td>
</tr>
<tr>
<td>19</td>
<td>COUNTER = COUNTER * INDEX</td>
</tr>
<tr>
<td>20</td>
<td>A(B(3))=A(3)</td>
</tr>
</tbody>
</table>
Table 4.2: Extracted parallelising information from the example FORTRAN 77 source code.

<table>
<thead>
<tr>
<th>Line</th>
<th>Nesting level</th>
<th>DEF</th>
<th>#</th>
<th>USE</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>{A(5)}</td>
<td>1</td>
<td>{A(4),B(3)}</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>{COUNTER}</td>
<td>1</td>
<td>{COUNTER}</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>{INDEX}</td>
<td>1</td>
<td>{INDEX}</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>{I}</td>
<td>1</td>
<td>{N}</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>{C(I)}</td>
<td>1</td>
<td>{I,X,Y}</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>{}</td>
<td>0</td>
<td>{}</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>{I}</td>
<td>1</td>
<td>{H}</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>{B(I)}</td>
<td>1</td>
<td>{I,M(I),X}</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>{}</td>
<td>0</td>
<td>{}</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>{I}</td>
<td>1</td>
<td>{N}</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>{A(I)}</td>
<td>1</td>
<td>{I,X,A(I)}</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>{}</td>
<td>0</td>
<td>{}</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>{B(3)}</td>
<td>1</td>
<td>{B(0),A(0)}</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>{I}</td>
<td>1</td>
<td>{N}</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>{J}</td>
<td>1</td>
<td>{N}</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>{D(I,J)}</td>
<td>1</td>
<td>{I,J,D(I+1,J),D(I,J+1),INDEX}</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>{}</td>
<td>0</td>
<td>{}</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>{}</td>
<td>0</td>
<td>{}</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>{COUNTER}</td>
<td>1</td>
<td>{COUNTER,INDEX}</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>{A(B(3))}</td>
<td>1</td>
<td>{B(3),A(3)}</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 4.3: Extracted specific loop information from the example FORTRAN 77 source code.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Nesting level</th>
<th>Iteration variable</th>
<th>Start value</th>
<th>End value</th>
<th>Increment value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-6</td>
<td>0</td>
<td>I</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>7-9</td>
<td>0</td>
<td>I</td>
<td>1</td>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>10-12</td>
<td>0</td>
<td>I</td>
<td>1</td>
<td>N+1</td>
<td>1</td>
</tr>
<tr>
<td>14-18</td>
<td>0</td>
<td>I</td>
<td>2</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>15-17</td>
<td>1</td>
<td>J</td>
<td>2</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

For each line of code shown in Table 4.1, the DEF and USE sets have to be computed, as well as the number of elements of each set, as illustrated in Table 4.2; for example, in line 1, A(5) is a modified variable and is therefore part of the DEF set, whereas A(4) and B(3) are used variables, and are listed in the USE set. Also, for each loop from the original code, the specified loop information needs to be extracted; this information, and the lines of code to which it refers, is shown in Table 4.3.

4.4 Conclusion

In order to guide the reader towards the development of Papa, this chapter has provided a justification for certain decisions taken, such as the choice of Fortran as the language of the original source code, and the choice of a standard from all the existing Fortran standards. The reasons of such a choice have also been explained.

The problems with the analysis of Fortran code have then been highlighted, and the exact information to be extracted from that code has been detailed and exemplified.
Chapter 5

Previous work

Much work has been done in the field of automatic parallelisation, and many tools have been used to extract some form of parallelising information from existing source code. Rather than providing a full listing of all existing tools, this section focuses on those which provide an internal representation of that information, and which could be used to the same effect as Papa.

The descriptions provided are intentionally kept concise and clear, and concentrate on the features available that could provide the information required by Paragen; more detailed explanations about these and other tools are available in the related bibliography, including comparison tests between several of the mentioned systems (Chaudhary et al., 1996; McKinley et al., 1995; Wilson et al., 1993).

5.1 Existing tools

5.1.1 Parafrase-2

The Parafrase-2 compiler (Polychronopoulos et al., 1989) is a source-to-source, multi-language restructuring tool, developed at the University of Illinois at Urbana-Champaign. It accepts source programs from a number of different programming languages as input, and likewise is able to produce its output using any one of those languages. This is achieved through the use
of an internal representation, that emphasises data and control dependencies rather than source language syntax. The use of different input and output languages is therefore solely dependent on the use of pre- and post-processors for the desired languages; the pre- and post-processors for C, FORTRAN 77 and Cedar Fortran (Hoeflinger, 1991) are available, allowing both serial and parallel code to be analysed. Figure 5.1 illustrates this process.

Although a parallel compiler on its own, Parafrase has been used on many systems (Poulsen & Yew, 1994; Su et al., 1995) as a front end analyser to their own parallel compilers. This is due to the broad variety of analyses it provides, which include base analyses (data and control dependencies), necessary for handling parallelism, as well as symbolic analysis and inter-procedural analysis; this provides tools with a wide range of information, including flow, dependence and call graphs.

### 5.1.2 Polaris

Also from the University of Illinois comes Polaris (Blume et al., 1996), another source-to-source optimising translator. The objective of the system is the automatic explicit parallelisation of sequential FORTRAN 77 code for a variety of architectures and machines; so far, shared-memory architectures
have been targeted, but projects are underway for targeting multi-processor workstations and scalable, shared-memory multi-processor architectures.

The output of the tool consists of FORTRAN 77 code, augmented with annotations indicating parallelism for several dialects of parallel Fortran; often Polaris will leave the actual transforming of code to platform-specific, back-end compilers. Figure 5.2 gives an overview of the Polaris approach.

The implementation of Polaris consists of 170,000 lines of C++ code; a basic infrastructure provides a hierarchy of C++ classes that developers can use for manipulating and analysing the input program.

Polaris uses an abstract syntax tree as its internal representation (Faigin et al., 1994) of a Fortran program. Through the use of an object-oriented approach, a program is divided onto classes of structures, from a program class to a statement class. Although somewhat complex, this object-oriented internal representation makes it easy to adapt Polaris to other source-code languages, although some changes to its consistency checks and analysis routines may be required.
5.1.3 ParaScope

ParaScope (Cooper et al., 1993) is an interactive parallel programming environment from Rice University, developed to support scientific programming of shared-memory multi-processor architectures. It accepts Fortran-D (Fox et al., 1990; Hiranandani et al., 1992) source code as input, and again produces Fortran-D source code as output.

The user can specify which procedures to parallelise through the composition editor, and can interact with the parallelising process through the ParaScope editor. Using this information as input, the program compiler generates the output parallelised code; the use of both a program and a module compiler allows the system to perform inter-procedural analysis, improving the parallelisation level of the output code. Figure 5.3 illustrates the interaction of the various components of the system.

The internal representation used is based on an abstract syntax tree, from which the parallel information can be extracted. Unfortunately, it is difficult to modify or extend (McKinley et al., 1995), because it is exposed to the entire system, so extending the internal representation to support new language extensions requires modifications throughout ParaScope.
5.1.4 SUIF

The SUIF (Stanford University Intermediate Format) compiler framework (Wilson et al., 1994; Hall et al., 1996) is designed for research in compilation techniques, especially automatic parallelisation. It consists of a small, clearly documented kernel, and a toolkit of compiler optimisers built on top of the kernel. The kernel defines the intermediate representation, and provides functions to access and manipulate that representation. The toolkit includes programming language front-ends, a set of parallel optimisers, and a set of compiler development tools. The structure of the SUIF framework is illustrated in Figure 5.4.

The SUIF compiler currently accepts Fortran or C source code, but uses a modified version of f2c (Feldman et al., 1990) to convert Fortran code to C code prior to analysis. According to the authors (Wilson et al., 1994), the conversion from Fortran to C occasionally obscures the high-level program semantics originally available in Fortran programs, and the use of a Fortran front-end would be ideal.

The SUIF format provides two levels of abstraction: at the higher level, it represents language constructs such as loops, array references and if state-
ments, and at the lower level it represents the remaining constructs, in a low level form resembling that of RISC instructions. This enables the extraction of information aimed at both levels of abstraction, and facilitates the interleaving of loop-level and low-level optimisations.

Much work has been done recently on the new SUIF2 compiler (Aigner \textit{et al.}, 2000); the original SUIF implementation (referred to as SUIF1) was designed to support high-level program analysis of C and Fortran programs, whereas the new SUIF2 format is aimed at a broader range of programming languages and paradigms (including object-oriented programming languages), and the structure of the framework in which it is integrated provides a more modular system, that enables components to inter-operate in a flexible manner. The new framework provides conversion tools which ensure backwards compatibility for older SUIF1 representations.

\section{VFCS}

VFCS (Vienna Fortran Compilation System) (Chapman \textit{et al.}, 1992; Benkner \textit{et al.}, 1995) is an interactive, source-to-source translator from the University of Vienna. It accepts Vienna Fortran, FORTRAN 77, HPF and Fortran 90 as input, and generates explicitly parallel programs, based on the SPMD paradigm.

VFCS operates in both interactive or batch modes. The batch mode allows for the automatic application of parallel analysis and construction of the explicitly parallel output program, through the use of a special batch command language; in the interactive mode, users have access to a set of analyses and a catalog of transformations via a graphical user interface.

VFCS uses abstract syntax trees for intermediate representation, and a program database for communication among components of the compiler; the program database contains syntax trees, call graphs, inter-procedural information, dependence graphs, and data partitioning information. Optimisations work directly on the program database, and transformations are guided by an interactive kernel.
The objective of VFCS is to generate optimised code for distributed-memory machines, and its internal representation is aimed towards this use; it is not readily adaptable to other languages, because the optimisations are too tightly coupled with the rest of the system (McKinley et al., 1995).

5.2 Why develop an entirely new tool

As there are so many tools available that extract parallelising information from FORTRAN 77 code, one could expect to use one of those to extract the information required by Paragen. There are however certain issues concerning the tools presented:

- Many of them present information in their proprietary format: this format tends to be coupled with the tool, and often changes with new versions of that tool (with the exception of SUIF, which has instead concentrated on a format, and built a whole environment around it).

- The information extracted is very complete, containing data dependencies on all tools, and flow dependencies on most, whereas Paragen only requires a fraction of this information.

- The information presented by some of the tools is aimed at specific hardware\(^1\), whereas Paragen is a more general tool.

The approach taken in the SCARE project, particularly in the Paragen tool, is one of simplicity: only strictly necessary information (detailed in Section 4.3.2) should be handled, in an intentional effort to keep the process of evolution of transformations not only simple, but also unbiased towards specific transformations, which might be useful on certain code or architecture, but that may prevent a better parallelisation of different code.

If the internal representation of one of the tools presented was to be used, then a grammar describing that representation would have to be written, and

\(^1\)SUIF has solved this problem in the new SUIF2 format.
a filter tool designed, which would use that grammar to filter out the information not needed by Paragen; this would result in wasted computational time, as the extraction of that information would not be needed in the first place. This approach also presents the problem of the selected tool changing its internal representation format, which, depending on the extent of that change, could mean a complete redesign of the filter tool built on top of it.

It was therefore deemed more appropriate to develop a new tool, using a standard FORTRAN 77 grammar, which can be extended to incorporate extensions to the language (such as the ones detailed in Section 7.1.2).

As mentioned in Section 3.3, there exists already a pre-processor for the C language, developed for the SCARE project, but adapting it to work with Fortran would be a daunting task, such is the difference between the constructs and data structures of the two languages. Another possibility would be to use software such as f2c to translate the source code from Fortran to C, and then use the existing C pre-processor; however, for reasons already explained, the use of f2c is not recommended, and as the existing tool is very limited in what concerns the information extracted, it made more sense to develop an entirely new pre-processor specific to Fortran from the start.

5.3 Conclusion

The Fortran language was created over 45 years ago, and countless compilers have been created for its various flavours; furthermore, the effort to create parallel programming compilers or environments has led to the creation of many tools that extract parallelising information from Fortran source code.

Many of these tools are extremely complete; they extract most of the data required for the parallelisation of the source code, often after an extensive analysis of that code. Paragen, however, needs only a fraction of that information, as it takes a completely different approach to the task of parallelising code. The development of a lighter tool, which can be incorporated on the SCARE project as its proprietary code, was therefore deemed necessary.
Chapter 6

Practical issues

The complexity of most legacy code, the reason that motivated the search for automated processes for its re-engineering in the first place, also has to be taken into consideration when using those automated processes.

In the specific case of Papa, there’s much more to its function than just extracting sets of used and modified variables from each statement in a legacy FORTRAN 77 source code, as outlined in Chapter 4. For example, a function reference in a line of the original serial code raises questions about used/modified parameters, usage of global variables, etc. There are also other problems, such as ensuring the sequencing of instructions from the FORTRAN 77 standard is not violated by Paragen’s transformations, and other instruction specific problems.

This chapter looks at all these issues, describing the problems detected, and the solutions found for those problems. It starts by describing the problems arising from the usage of different program units, then moves on to describing the labelling techniques used to solve statement sequencing problems, and finally describes other problems found with specific instructions, along with the solutions employed.
6.1 Program unit inter-communication

6.1.1 Description

As with most programming languages, FORTRAN 77 programs are often broken down into several procedures.\(^1\) There are four kinds of procedures defined in the FORTRAN 77 standard: intrinsic functions, statement functions, external functions, and subroutines. These types of procedures are classified according to the diagram shown in Figure 6.1, and are briefly described below.

Intrinsic functions

The FORTRAN 77 standard specifies that all FORTRAN 77 compilers must supply certain built-in functions, known as intrinsic functions. These functions consist of a wide range of applications, including type conversion, array search, string manipulation, trigonometric functions, and many more.

\(^1\)The notion of procedure used here is that of a section of code referenced (or called into operation) from another part of the program.
**Statement functions**

In FORTRAN 77, a statement function statement may be used to define a new function in a program unit, provided that the function is so simple that it can be defined using a single statement, and that the function will be referenced from one program unit only.

For example, the following line of code defines a function called QUAD, with two arguments \( X \) and \( Y \):

\[
\text{QUAD}(X,Y) = A \times X^2 + B \times X \times Y + C \times Y^2
\]

This function, if defined at the beginning of the current program unit, can be referenced like any other function, but only inside the current program unit.

**External functions and subroutines**

An external function or a subroutine are program units, usually written by the authors of the FORTRAN 77 source code, which start with a `FUNCTION` statement or a `SUBROUTINE` statement, respectively; to execute an external function, a reference to its name is made on a statement, whereas to execute a subroutine, the `CALL` statement is used. For example, if `FUNX` was a function defined in a program unit, and `SUBX` a subroutine defined on another program unit, the following two statements are valid (assuming the number and type of the arguments is correct):

\[
\begin{align*}
X &= \text{FUNX}(Y) \\
\text{CALL} & \text{ SUBX}(Y)
\end{align*}
\]

External functions and subroutines can also be defined on shared libraries, in which case they may not be written in FORTRAN 77, but their usage on a FORTRAN 77 program is the same (the difference being that they will only be provided to the compiler at linkage time).
6.1.2 Referencing procedures

As mentioned before (Section 3.2), Paragen works on a program unit at a time. This means that Papa will be given as input the text of a single program unit (be it the main program, a function, or a subroutine).

For example, if a program is composed of a main program called MAIN, and has five functions called A, B, C, D and E, according to the call graph shown in Figure 6.2, each of these program units will have to be passed to Papa, one at a time.

When parsing one of those program units, Papa is likely to come across a reference to one of the five functions. If this function is unknown (i.e., if it has not been parsed yet), then no information is available regarding its code, and so a worst case scenario is assumed, which means that all parameters are assumed to be used and modified, and similarly all global variables are assumed to be used and modified.
It is therefore essential to call **Papa** according to the call graph of the serial program to parallelise. For the given example, if the decision is made to parallelise function B, then **Papa** must first be applied to function E (so that information is extracted about used and modified parameters and global variables), then to function D, and finally to the file containing function C, before parsing function B.

In any case, when a reference to a program unit not previously parsed by **Papa** is made, such as reference to an intrinsic function or a library function, information must be provided about these functions in what concerns used and modified parameters and global variables. In the case of the standard FORTRAN 77 intrinsic functions, **Papa** contains in-built information about these; in the case of unknown library functions (or as yet unparsed program units), for which no information has been provided, these will be considered unknown references, and the worst case scenario described above will be assumed.

### 6.1.3 Functions as arguments

FORTRAN 77 provides a way to pass function pointers as arguments to a function or subroutine. Although a powerful feature, it has the downside of creating a complex control flow on the program, making it often impossible to draw a call graph. For example, in the following subroutine declaration:

```fortran
SUBROUTINE APPLY(F,X,Y)
  Y=F(X)
END
```

the first argument is a function pointer, and the return value of the subroutine depends on which function F is pointing to. When parsing this subroutine, **Papa** has no way of knowing which function has been passed as an argument; as a result, the worst case scenario is assumed, and all arguments passed to the function (in this case, the X variable) are considered used and modified,
as well as all global variables, and this information will be passed down to the original reference to the program unit. So, in a reference to the subroutine APPLY like this:

\[
\text{CALL APPLY(FUNX, X, Y)}
\]

the \textit{DEF} set will include the variables \{X,Y\}, and the \textit{USE} set will contain the variable \{X\} (but not FUNX, as FUNX is a function name, not a variable)\(^2\), and all global variables will also be declared as being both used and modified.

\section*{6.2 Line sequencing problems}

As described previously (Section 3.2), Paragen is free to exchange or execute in parallel two statements from the original serial code, as long as there are no data dependencies between those two statements. However, depending on the programming language used, certain instructions might not be freely exchangeable, either due to specific guidelines concerning statement sequencing from the programming language used, or due to the specific nature of those instructions. In such cases, it is up to \textbf{Papa} to prevent those statements from being moved around freely, in a manner which is transparent to Paragen. The technique used is described in this section, along with the cases where its application is needed.

\subsection*{6.2.1 Fortran statement sequencing}

\textbf{Description}

The FORTRAN 77 standard has very strict guidelines concerning the sequencing of statements in a program unit, shown in Table 6.1. This table shows categories of statements that can be interspersed (i.e., executed within

\(^2\)How the symbol FUNX is detected as being a function name and not a variable is done through a symbol table, and will be explained in Chapter 7.
Table 6.1: Required order of statements of various types and comment lines in a program unit, according to FORTRAN 77 standard.

<table>
<thead>
<tr>
<th>Comment lines</th>
<th>Program, Function, Subroutine, or Block Data statement</th>
<th>IMPLICIT parameter statements</th>
<th>Other specification statements</th>
<th>Statement function statements</th>
<th>Executable statements</th>
<th>END statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY statements</td>
<td>FORMAT statements</td>
<td>PARAMETER statements</td>
<td>Data statements</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each other) separated by vertical lines, whereas horizontal lines define categories of statements that must not be mixed, and must be executed in the order of execution shown. More specifically:

- the first statement in a program unit is either a PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement;

- PARAMETER statements may be interspersed with both IMPLICIT statements and all other specification statements;

- IMPLICIT statements must precede all other specification statements (except PARAMETER statements);

- all specification statements (including PARAMETER and IMPLICIT statements) must precede all executable statements (including DATA statements and statement function statements);

- DATA statements may be interspersed with both statement function statements and executable statements;
• statement function statements must precede executable statements;
• FORMAT statements may occur anywhere within a program unit;
• ENTRY statements may occur anywhere within a program unit, except between an IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO loop;
• the last statement of a program unit must be an END statement;
• comment lines may occur anywhere in a program unit (even before the first statement), but before the END statement.

Problems and solution

To ensure the correct sequencing of statements, a set of neutral statements has been introduced, shown in Table 6.2. The objective of these neutral statements is to create dependencies that will in turn create boundaries between statements. To achieve this, a set of variables (1st, 2nd and 3rd) is used, to introduce specific dependencies that maintain the sequencing of statements set by the FORTRAN 77 standard.3

Exactly how this works is best shown with an example. Consider the following extract of a FORTRAN 77 source code:

```
PROGRAM SIMPLE
  INTEGER W, X, Y
  FUNX(Z)=3.14*Z
  Y=2
  X=FUNX(Y)
  W=Y**2
  . . .
```

As variable names in FORTRAN 77 are a sequence of six capital letters or numbers, starting with a letter (American National Standards Institute, 1978), the name of the variables introduced was chosen to avoid a conflict with possibly existing variable names.

3
Table 6.2: Solution for the problem of keeping the order of statements as specified in FORTRAN 77 standard.

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>USE</th>
<th>DEF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA st.</strong></td>
<td><code>{}</code></td>
<td><code>{1st}</code></td>
</tr>
<tr>
<td><strong>PARAMETER statements</strong></td>
<td><code>{1st,2nd}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><strong>IMPLIED statements</strong></td>
<td><code>{1st,3rd}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><strong>ENTRY statements</strong></td>
<td><code>{1st}</code></td>
<td><code>{2nd,3rd}</code></td>
</tr>
<tr>
<td><strong>DATA statements</strong></td>
<td><code>{1st,2nd}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><strong>EXECUTABLE statements</strong></td>
<td><code>{1st,3rd}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><strong>END statement</strong></td>
<td><code>{}</code></td>
<td><code>{1st}</code></td>
</tr>
</tbody>
</table>
As there is no data dependency between the third and fourth statements, Paragen can make the decision of executing them in a different order. Although semantically correct from Paragen’s point of view (as it does not introduce a data dependency), this would create an invalid sequence of instructions, as it does not comply to the FORTRAN 77 specification for instruction sequencing.

With the use of neutral instructions, Papa inserts a separation line between the specification statement declaring the variables W, X and Y, and the statement function statement that follows it, and another separation line between that statement function statement and the three executable statements that follow it. The resulting DEF and USE sets for these statements are those shown in Table 6.3.

Table 6.3: Applying the labelling technique to a sequence of FORTRAN 77 sentences.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>DEF</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM SIMPLE</td>
<td>{1st}</td>
<td>{}</td>
</tr>
<tr>
<td>INTEGER W,X,Y</td>
<td>{W,X,Y}</td>
<td>{1st,3rd}</td>
</tr>
<tr>
<td>(separation line)</td>
<td>{2nd,3rd}</td>
<td>{1st}</td>
</tr>
<tr>
<td>FUNX(Z)=3.14*Z</td>
<td>{}</td>
<td>{1st,3rd}</td>
</tr>
<tr>
<td>(separation line)</td>
<td>{3rd}</td>
<td>{1st}</td>
</tr>
<tr>
<td>Y=2</td>
<td>{Y}</td>
<td>{1st,3rd}</td>
</tr>
<tr>
<td>X=FUNX(Y)</td>
<td>{X}</td>
<td>{1st,3rd,Y}</td>
</tr>
<tr>
<td>W=Y**2</td>
<td>{W}</td>
<td>{1st,3rd,Y}</td>
</tr>
</tbody>
</table>

This forces Paragen to keep the FORTRAN 77 sequencing of statements, without specific knowledge of this sequencing, thus preserving Paragen’s language independence. The resulting information allows Paragen to exchange only the last two statements (as both use variables 1st, 3rd, and Y, but none modifies them), or to execute them in parallel.
The use of these neutral instructions has the downside of introducing further dependencies for Paragen to compute, so these instructions are only used if needed. For example, in a function or subroutine with no IMPLICIT statements, the neutral instruction dividing those instructions from other specification statements should not be inserted; similarly, in a function or subroutine with no statement functions, the line used to separate these from executable statements should not be inserted.

As mentioned previously, other restrictions exist, such as that ENTRY statements must not occur between a block IF statement and the corresponding END IF statement, or between a DO statement and the terminal statement of its DO-loop. But since Paragen never exchanges lines of code with different nesting levels, and statements inside IF and DO blocks have a higher nesting level than the statements outside them, Papa does not have to worry about this sequencing restriction.

6.2.2 GO TO statements

Ever since the publication of Dijkstra’s famous letter (Dijkstra, 1968), where he states that “the quality of programmers is a decreasing function of the density of go to statements in the programs they produce”, GO TO statements became the center of an intense debate between those that swore against its use, and those that defended its use under certain restrained circumstances (Knuth, 1974). Nowadays, programs using GO TO statements are mostly looked upon as bad programming examples, and modern languages do not support its use (the GO TO statement is not supported in Java, even though the keyword goto is reserved). In the 1960s, however, when high-level languages were still emerging, many programmers used the GO TO statement, either because of lack of better control statements in their language of choice, or as a result of years of experience in programming in low-level languages (such as machine language).

4The case statement, for example, was first introduced by the Pascal language, in 1972, and was not introduced in Fortran until the Fortran 90 standard.
The misuse of \texttt{GO TO} statements is often a sign of poor design and unstructured thinking. The problem is that it is an easy statement to use, and is often a convenient and fast means of overcoming a tricky piece of coding.

In FORTRAN 77, it is sometimes difficult to completely avoid using the \texttt{GO TO} statement. The following list illustrates some cases where it can be used sensibly:

- when implementing the behaviour of a \texttt{loop while} statement;
- when implementing the behaviour of a \texttt{case} statement;
- in error situations, where control must pass from the main algorithm to a special terminating or recovery sequence;
- in circumstances where it is necessary to jump out of a loop cycle.

The FORTRAN 77 standard defines three types of \texttt{GO TO} statements, which are described in the next sections, as well as the problems they create for \texttt{Papa}, and the techniques used to eliminate those problems are presented.

\textbf{Unconditional GO TO statement}

This is the simplest form of a \texttt{GO TO} statement; it has the form:

\begin{verbatim}
GO TO \texttt{s}
\end{verbatim}

where \texttt{s} is a statement label of an executable statement that appears in the same program unit as the unconditional \texttt{GO TO} statement; its execution transfers control to the statement identified by the label.

When a program contains this kind of \texttt{GO TO} statement, certain precautions have to be taken concerning the evolution of transformations by Paragen. For example, take the FORTRAN 77 subroutine shown in Table 6.4, that fills the top right half of an $M \times N$ array with a value specified by a parameter. Since Paragen looks only at variable dependencies when evolving transformations, it can decide that lines 4 and 5 are to be exchanged, as both use variable \texttt{I} but neither modifies it. However, that transformation
will change the semantic meaning of the program, even though no dependencies have been broken, as seen in the code shown in Table 6.5: the variable J will only be updated once during the execution of the code, meaning that only the first line of the array will be initialised to the value specified.

Table 6.4: Example program using unconditional *GO TO* statements.

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SUBROUTINE TOPR(K,M,N,V)</td>
</tr>
<tr>
<td>2</td>
<td>INTEGER K(M,N),V,I,J</td>
</tr>
<tr>
<td>3</td>
<td>I=1</td>
</tr>
<tr>
<td>4</td>
<td>15 IF(I.GT.M) GO TO 35</td>
</tr>
<tr>
<td>5</td>
<td>J=I</td>
</tr>
<tr>
<td>6</td>
<td>25 IF(J.GT.N) THEN</td>
</tr>
<tr>
<td>7</td>
<td>I=I+1</td>
</tr>
<tr>
<td>8</td>
<td>GO TO 15</td>
</tr>
<tr>
<td>9</td>
<td>ELSE</td>
</tr>
<tr>
<td>10</td>
<td>K(I,J)=V</td>
</tr>
<tr>
<td>11</td>
<td>J=J+1</td>
</tr>
<tr>
<td>12</td>
<td>GO TO 25</td>
</tr>
<tr>
<td>13</td>
<td>END IF</td>
</tr>
<tr>
<td>14</td>
<td>35 END</td>
</tr>
</tbody>
</table>

To solve this problem, a technique similar to that used to ensure the correct sequencing of instructions is used; in this case, the value stated as being the used or modified variable will be the value of the statement labels used by *GO TO* statements, or attached to executable statements (such as the numbers 15 and 35 on line 5), concatenated by the letter 'f'.\footnote{Since a statement label is a sequence of up to five digits with at least one of them being nonzero (American National Standards Institute, 1978), these sequences of numbers could be mistaken by unit specifiers (described in Section 6.3.5), hence the concatenation of the letter 'f' on the technique now being presented.}
Table 6.5: How the exchange of two statements with no dependencies between them can cause semantic changes in a program, if GO TO statements are used.

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SUBROUTINE TOPR(K,M,N,V)</td>
</tr>
<tr>
<td>2</td>
<td>INTEGER K(M,N),V,I,J</td>
</tr>
<tr>
<td>3</td>
<td>I=1</td>
</tr>
<tr>
<td>4</td>
<td>J=I</td>
</tr>
<tr>
<td>5</td>
<td>15 IF(I.GT.M) GO TO 35</td>
</tr>
<tr>
<td>6</td>
<td>25 IF(J.GT.N) THEN</td>
</tr>
<tr>
<td>7</td>
<td>I=I+1</td>
</tr>
<tr>
<td>8</td>
<td>GO TO 15</td>
</tr>
<tr>
<td>9</td>
<td>ELSE</td>
</tr>
<tr>
<td>10</td>
<td>K(I,J)=V</td>
</tr>
<tr>
<td>11</td>
<td>J=J+1</td>
</tr>
<tr>
<td>12</td>
<td>GO TO 25</td>
</tr>
<tr>
<td>13</td>
<td>END IF</td>
</tr>
<tr>
<td>14</td>
<td>35 END</td>
</tr>
</tbody>
</table>

For the specific case of unconditional GO TO statements, the following three rules are applied:

- every time an executable statement is preceded by a statement label, the value of that statement label is declared as being a modified variable;

- every time a line has an unconditional GO TO statement followed by a statement label, the value of that statement label is declared as being a modified variable;

- every executable statement will be reported as using all the labels that are referenced by GO TO statements in the program.
In the previous example, this results in the \textit{DEF} and \textit{USE} sets seen in Table 6.6 (that does not include the sequencing labels 1st, 2nd and 3rd for the sake of clarity).

Table 6.6: Modifying the \textit{DEF} and \textit{USE} sets when unconditional \texttt{GO TO} statements are used.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>DEF</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SUBROUTINE TOPR(K,M,N,V)}</td>
<td>{}</td>
<td>{M,N}</td>
</tr>
<tr>
<td>\texttt{INTEGER K(M,N),V,I,J}</td>
<td>{K,V,I,J}</td>
<td>{M,N}</td>
</tr>
<tr>
<td>I=1</td>
<td>{I}</td>
<td>{15f,25f,35f}</td>
</tr>
<tr>
<td>15 IF(I.GT.M) \texttt{GO TO 35}</td>
<td>{15f,35f}</td>
<td>{15f,25f,35f}</td>
</tr>
<tr>
<td>J=I</td>
<td>{J}</td>
<td>{15f,25f,35f,I}</td>
</tr>
<tr>
<td>25 IF(J.GT.N) THEN</td>
<td>{25f}</td>
<td>{15f,25f,35f,J,N}</td>
</tr>
<tr>
<td>I=I+1</td>
<td>{I}</td>
<td>{15f,25f,35f,I}</td>
</tr>
<tr>
<td>\texttt{GO TO 15}</td>
<td>{15f}</td>
<td>{15f,25f,35f}</td>
</tr>
<tr>
<td>ELSE</td>
<td>{}</td>
<td>{15f,25f,35f}</td>
</tr>
<tr>
<td>K(I,J)=V</td>
<td>{K(I,J)}</td>
<td>{15f,25f,35f,I,J}</td>
</tr>
<tr>
<td>J=J+1</td>
<td>{J}</td>
<td>{15f,25f,35f,J}</td>
</tr>
<tr>
<td>\texttt{GO TO 25}</td>
<td>{25f}</td>
<td>{15f,25f,35f}</td>
</tr>
<tr>
<td>END IF</td>
<td>{}</td>
<td>{15f,25f,35f}</td>
</tr>
<tr>
<td>35 END</td>
<td>{35f}</td>
<td>{15f,25f,35f}</td>
</tr>
</tbody>
</table>

This technique ensures that both unconditional \texttt{GO TO} statements and labeled instructions are never moved, as dependencies are introduced between those statements and all other executable statements in the program unit. This forces those statements to be executed sequentially, while instructions between these two kinds of statements are freely swapped or parallelised by Paragen, just as long as there are no data dependencies between them.
Computed GO TO statement

The computed GO TO statement has the form:

\[ \text{GO TO } (s_1, s_2, \ldots, s_n), i \]

where \((s_1, s_2, \ldots, s_n)\) are statement labels, not necessarily different, attached to executable statements in the same program unit, and \(i\) is an integer expression (the comma preceding it is optional); the effect of this statement is as follows:

- if \(i = 1\), transfer control to the statement with statement label \(s_1\)
- if \(i = 2\), transfer control to the statement with statement label \(s_2\)
  
  ...

- if \(i = n\), transfer control to the statement with statement label \(s_n\)

Finally, if the value of \(i\) is less than 1 or greater than \(n\), no action is performed, i.e., the computed GO TO statement has the same effect as a CONTINUE statement.

This is a more complex statement, in that the instruction to which the control of the program is going to be sent might not be known before runtime (the value of \(i\) might be set depending on user input, for example). The solution for this kind of GO TO statement is however very similar to that used with unconditional GO TO statements, and uses the following rules:

- every time an executable statement is preceded by a statement label, the value of that statement label is declared as being a modified variable;

- every time a line has a computed GO TO statement followed by a list of statement labels, the values of those statement labels are declared as being modified variables;

- every line will be reported as using all the labels that are referenced by GO TO statements in the program.
Assigned GO TO statement

The assigned GO TO statement is a complicated statement, introduced in early FORTRAN standards for lack of better control instructions, and its use is strongly discouraged.\(^6\) It has two possible formats; in its simplest form, it is used as

\[
\text{GO TO } i
\]

where \(i\) is an integer expression having as its value the value of a statement label attached to some executable statement in the same program unit; when executed, this instruction transfers the control to the executable statement labeled by that statement label.

The second form is somewhat similar in form and behaviour to the computed GO TO statement; it has the form:

\[
\text{GO TO } i, (s_1, s_2, \ldots, s_n)
\]

where \((s_1, s_2, \ldots, s_n)\) are statement labels attached to executable statements in the same program unit, and \(i\) is an integer expression (the comma following it is optional); at the time of execution of such an assigned GO TO statement, \(i\) must have as its value the value of a statement label attached to some executable statement in the same program unit, and that statement label must be the value of one of the statement labels in the parentheses. Control is then transferred to that executable statement.

When using assigned GO TO statements, in order to assign an appropriate statement label to \(i\), the ASSIGN statement must be used; it has the form:

\[
\text{ASSIGN } s \text{ TO } i
\]

where \(s\) is a statement label attached to some executable or FORMAT statement in the same program unit.

\(^6\)The assigned GO TO statement, and the related ASSIGN statement, have existed in later Fortran standards for the sole purpose of backwards compatibility with both the FORTRAN 66 and FORTRAN 77 standards, and are to be completely removed in the new Fortran 200x standard (Fortran Standards Technical Committee, 2002).
To deal with assigned \texttt{GO TO} statements, the following set of rules is used:

- every time an executable line is preceded by a statement label, the value of that statement label is declared as being a modified variable;

- every time a line has an \texttt{ASSIGN} statement, the value of the statement label being assigned is declared as being a modified variable;

- every time a line has an assigned \texttt{GO TO} statement used in its simplest form, all labels which have appeared in all \texttt{ASSIGN} statements in the same program unit (both before and after the current statement) are declared as being modified variables;

- every time a line has an assigned \texttt{GO TO} statement followed by a list of statement labels, the values of those statement labels are declared as being modified variables;

- every line will be reported as using all the labels which are referenced by \texttt{GO TO} and \texttt{ASSIGN} statements in the program.

Note that these sets of rules can be applied in sequence, i.e. if a program makes use of unconditional and assigned \texttt{GO TO} statements, then each set of rules will be applied to the whole program unit.

\textbf{Arithmetic IF statement}

The arithmetic \texttt{IF} statement is a “dangerous” statement (Balfour & Marwick, 1988), since its use tends to be coupled with the usage of \texttt{GO TO} statements, and causes the logic of the program unit in which it occurs to be difficult to follow, and even more difficult to debug. It has the following form:

$$\textbf{IF } ( \textit{arithmetic expression} ) s_1, s_2, s_3$$

where \textit{arithmetic expression} is an arithmetic expression of any type (except \texttt{COMPLEX}), and \(s_1, s_2, s_3\) are statement labels attached to executable statements in the same program unit (they need not be all different). The effect of this
statement is that the arithmetic expression is evaluated: if its value is below zero, the control is transferred to the statement with label $s_1$, if it is equal to zero the control is transferred to the statement with label $s_2$, and if it is greater than zero the control is transferred to the statement with label $s_3$.

The behaviour of this kind of statement is similar to that of a computed \texttt{GO TO} statement, in that the statement to which the control will be transferred depends on the value of an expression. Therefore the rules defined for that kind of statement were adopted:

- every time an executable statement is preceded by a statement label, that statement label is declared as being a modified variable;
- every time a line has an arithmetic \texttt{IF} statement followed by a list of statement labels, the values of those statement labels are declared as being modified variables;
- every line will be reported as using all the labels that are referenced by arithmetic \texttt{IF} statements in the program.

\section*{Conclusion}

\texttt{GO TO} statements are known to easily sever the fluidity and sequencing of programs; Paragen, however, depends on the logical sequencing of statements to apply its transformations. The techniques presented here, to deal with \texttt{GO TO} statements, succeed in preventing Paragen from evolving transformations that might change the semantic meaning of a program unit; they do so by introducing dependencies that prevent Paragen from moving these statements in relation to all other lines of code.

It should be noted, however, that this creates a very strict structure for the program being re-engineered, as statements are forced to stay in their place; in the program shown in Table 6.6, for example, the application of the technique presented has the effect of preventing the parallel execution of any statement. For reasons of realism, much higher gains would be achieved if the shown subroutine was changed to make use of \texttt{DO} statements, for example.
6.2.3 Non-movable statements

Description

There are certain statements in the FORTRAN 77 standard that, due to their nature, are required to stay in their exact place in the program unit, and cannot be executed in parallel with any other statements. These are:

**CONTINUE statements**: these are used to terminate DO loops, and have the effect of returning the control to the line containing the DO statement that originally referenced them.

**PAUSE statements**: these statements, when executed, cause the execution of the executable program in which they occur to be suspended; execution may only be resumed by intervention of an external source, usually through user action.

**STOP statements**: these statements, when executed, cause the execution of the executable program in which they occur to be terminated; they are executable statements, and therefore may appear anywhere in a FORTRAN 77 program.

**ENTRY statements**: these may be used to specify alternative entry points into a function or subroutine program (ENTRY statements are discussed in more detail in Section 6.3.4).

**RETURN statements**: these statements, when executed, terminate the execution of the function or subroutine in which they appear, and return control to the referencing program unit.

**ELSE statements**: these statements are used when coding an *if-then-else* decision statement; they are non-executable statements, and have the effect of defining the start of an *ELSE-block*, defined to be all following statements up to, but not including, a corresponding **END IF** statement.
END IF statements: execution of these statements has no effect, and execution continues with the statement following them; they exist only for closing a IF statement (that is, for each IF statement, there must be a corresponding END IF statement).

Problems and solution

These statements control the flow of control on a program unit, and therefore should not be moved from their place; it is easy to imagine a situation where moving one of these statements would drastically change the semantic meaning of the program unit in which they occur.

To prevent Paragen from moving these statements, the variable 1st, introduced in Section 6.2.1, will be declared as being a modified variable in each line of code containing the four statements described here. Since Papa reports all statements to use the variable 1st, Paragen will not attempt to move any statement modifying that variable, so as not to break the data dependence between those statements and all other statements in the program unit; for the same reason, the statements modifying the variable 1st will not be executed in parallel.

IF statements should not be treated as non-movable statements, as Paragen can decide to parallelise the execution of an if-block with any other statement (or group of statements). Since Paragen will never exchange statements with different nesting levels, the ELSE and END IF statements inside a if-block are in a fixed position only inside that block; the IF statement, however, has the same nesting level as the other statements surrounding the if-block, and shouldn’t therefore be fixed, as that would prevent the execution of the whole if-block associated with it in parallel with other statements.

It should be noted that the solution presented here could be used with GO TO statements, as they are effectively treated as non-movable statements when using the technique presented in Section 6.2.2; however, the solution presented in that section could be of use to other researchers when only GO TO statements are being handled, which is why it is presented and used.
6.3 Other problems

Apart from sequencing and control flow problems, there are other specific problems concerning the information to be provided to Paragen by Papa. These range from syntactic problems to problems resulting from the semantic meaning of certain statements, and are all described in this section, along with examples illustrating them.\footnote{Please note that some of the examples shown are based on the referenced bibliography (Balfour & Marwick, 1988; Page, 1995).}

6.3.1 Arrays

Array referencing

As mentioned earlier (Section 4.2.3), in Fortran the syntax for an array reference is the same as for a function call, due to Fortran’s limited character set (square brackets are not part of it); this means that Papa needs to keep a symbol table that lists which arrays were declared, to make the distinction between array references and function references. For example, in the following statement:

\[
X = Y(Z)
\]

the DEF and USE sets cannot be calculated without prior knowledge of what the symbol Y represents. If Y is a declared array, then the DEF set will contain the variable \(\{X\}\), and the USE set will contain the variables \(\{Z, Y(Z)\}\); if Y is a function, however, the DEF and USE sets will depend on the usage of parameters (and global variables) inside the function.

Array element definition

For a similar problem as above, it is sometimes difficult to differentiate between the assignment of a value to an element of an array, and a statement
function statement. For example, in the following statement:

\[ Y(Z) = X \]

if \( Y \) is an array declared previously, then this is an executable statement, and therefore the \( DEF \) set will contain the variable \{\( Y(Z) \}\}, and the \( USE \) set will contain the variables \{\( Z, X \)\}; if \( Y \) has not been declared as an array, however, and this statement appears before any executable statements, then this is a statement function statement, declaring a function \( Y \) that takes one argument (labelled \( Z \)) and returns the current value of the variable \( X \).

As before, the use of a symbol table ensures the correct \( DEF \) and \( USE \) sets are reported for this statement, as \textbf{Papa} will be able to tell if \( Y \) has been declared as an array previously or not.

**Arrays as arguments to a procedure**

When an array is passed as an argument to a function or subroutine, and that function or subroutine uses or modifies any element of the array, then the statement referencing the function or subroutine will have its \( DEF \) and \( USE \) sets updated accordingly. For example, if a Fortran program contains the following code:

```fortran
INTEGER X(30)

... 
CALL SUBX(X,30)
```

where the arguments to the subroutine \texttt{SUBX} are an array and the number of its elements, if the subroutine uses both arguments, then the \( USE \) set for the line containing the \texttt{CALL} statement will contain the variable \{\( X \)\}.

There are cases, however, when only a section of an array should be passed to the subroutine. This is allowed in FORTRAN 77, but the syntax for it is somewhat confusing; for example, using subroutine \texttt{SUBX}, to pass only the
last 20 elements of array \( X \) as an argument, the statement to write would be:

\[
\text{CALL SUBX}(X(10), 20)
\]

This subroutine reference can be deceiving; if no previous knowledge of the subroutine \textbf{SUBX} is available, then it can be seen as either passing two \texttt{INTEGER} values as arguments (the value contained in index position 10 of array \( X \), and the value 20), or passing a sub-array of \( X \), starting on position 10, and its size, which is effectively the value 20.

Therefore the decision on whether declaring \( X \) or just \( X(10) \) to be the used argument of the subroutine call is dependent on the analysis of the subroutine, which means that when \textbf{Papa} analyses that subroutine, the data it extracts must contain information not only about which arguments are used and modified, but also on how those arguments are interpreted and used.

### 6.3.2 Common blocks

**Description**

Common blocks in \texttt{FORTRAN 77} work as shared zones of memory to which different program units may have access (in other words, the mechanism available in \texttt{FORTRAN 77} to declare global variables). A common block may have a name by which it is known in all program units using it, making it global to the whole program. There is one specific common block which has no name, and is referred to as the blank common block; it is available to all program units. A common block is specified by a \texttt{COMMON} statement, which has the following form:

\[
\text{COMMON} \  /\text{blockname}_1/\text{list}_1, /\text{blockname}_2/\text{list}_2, \ldots, /\text{blockname}_n/\text{list}_n/ \\
\]

where each \texttt{blockname} is the name of a common block, or if the name is omitted, the blank common block is assumed (if the first block name is omitted, the two slashes are optional). Each \texttt{list} is then a list of variable names, array
names and array declarators, separated by commas. The commas separating each `/blockname/list` couple are optional.

The `COMMON` statement declares that the entities in `list_i` are in the common block with name `blockname_i`. Those entities are held in the sequence in which they were declared. For a program unit to access a common block, it has to declare that common block, along with the names of entities that will be used inside that common block; since different program units might use different names for those entities, the names are local to each program in which the common block is specified, while the entities themselves are global (since they represent the physical location in the common memory block).

For example, let us consider the following three statements:

\[
\begin{align*}
\text{COMMON/INFO/I,K,AVG(2),Z} \\
\text{COMMON/INFO/K(3),/INFO/A(2)} \\
\text{COMMON/INFO/I,J,ALL,Y,Z}
\end{align*}
\]

Suppose these statements are contained in the main program and the subroutines `AVERAGE` and `DEVIATION`, respectively. They all refer to the same common block, `INFO`, which contains five entities. They create the name/entity mapping seen in Table 6.7. This means that, for example, if the variable `ALL` is modified in the subroutine `DEVIATION`, then the same change is applied to variable `AVG(1)` in the main program, and possibly to the third element of array `K` in subroutine `AVERAGE` (depending on whether or not subroutine `DEVIATION` was called from within subroutine `AVERAGE`).

According to the standard, the type of each entity remains the same for all program units, so all references to each entity must assume the same type (except that a complex value in one program unit may be referenced as two real values in another). Also, the size of a common block is the sum of the sizes of all the entities declared to be in that block, and common blocks with the same name, except for the blank common block, must have the same size.
Table 6.7: Example mapping from local names to global entities within a common block.

<table>
<thead>
<tr>
<th>INFO common block</th>
<th>entity1</th>
<th>entity2</th>
<th>entity3</th>
<th>entity4</th>
<th>entity5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main program</td>
<td>I</td>
<td>K</td>
<td>AVG(1)</td>
<td>AVG(2)</td>
<td>Z</td>
</tr>
<tr>
<td>Subroutine AVERAGE</td>
<td>K(1)</td>
<td>K(2)</td>
<td>K(3)</td>
<td>A(1)</td>
<td>A(2)</td>
</tr>
<tr>
<td>Subroutine DEVIATION</td>
<td>I</td>
<td>J</td>
<td>ALL</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

Apart from the name and the size differences, there are a few more differences between named common blocks and the blank common block. The entities of a named common block declared in a program unit will become undefined (that is, their values will not be saved for the next reference to that common block) on exit from that program unit, unless another program unit currently referencing this program unit has also specified the same common block, or the common block name appears in a `SAVE` statement. Since the main program is at the top of the call graph for all program units, the entities in a named common block declared in the main program will never become undefined. Entities in the blank common block will never become undefined regardless of where the block is declared.

Problems and solution

Programs with `COMMON` blocks are usually very hard to parallelise (Chapman et al., 1994); the sharing of common areas of memory makes parallelising procedures that use those areas a very difficult task.

In the specific case of **Papa**, when a program unit uses or modifies entities in a common block, Paragen needs to be informed about this fact. For example, taking the example shown in Table 6.7, if the variable **ALL** is modified in the **DEVIATION** subroutine, and this subroutine is called from the main program, then the normal output of **Papa** for the main program, in the line that references the **DEVIATION** subroutine, must be changed to inform Paragen that the variable **AVG(1)** has been modified. In other words, when
parsing a program unit that uses or modifies entities in a common block, Papa needs to save information about which entities on each common block declared are used and/or modified. That way, when parsing a program unit (such as the main program) that calls the DEVIATION subroutine, information will be available about used/changed entities on each common block.

There are two major problems with the proposed approach, however:

- the use of different types, on different program units, to access the same entities of a common block;
- the use of arrays within common blocks.

Although the standard specifies that entities should always be accessed by variables of the same type (and therefore of the same size) from all program units, in reality this is not followed by all compilers. The result of a program making use of this non-standard feature can however be impossible to predict, since different architectures have different sizes for different data types.

This means that a possible scenario is the one where a program has a non-standard use of common blocks, but produces acceptable output; the information produced by Papa, concerning that program, could however be incorrect, without specific knowledge of the hardware and compiler originally used.

For these reasons, the following approach to common blocks was adopted:

- if a program unit declaring a common block uses any of the entities in that common block, it is considered that all entities in that common block are used;
- if a program unit declaring a common block modifies any of the entities in that common block, it is considered that all entities in that common block are modified.

For example, g77, the GNU Fortran compiler, which has been used for most of the code testing during the development of Papa, accepts different types to access the same entities in a common block.
Taking the same example as before, this means that if variable ALL is modified in the DEVIATION subroutine, and this subroutine is referenced from the main program, then at the time of that reference, the variables I, J, AVG (the whole array) and Z are considered modified.

Although this is an approach that greatly reduces the level of parallelisation that can be applied to a program making use of common blocks, it guarantees that those transformations that will be applied will not change the semantic meaning of the original serial program. The use of a better approach would involve collecting information about the hardware for which the legacy code was originally designed for, as well as information about the compiler used; this is listed as a possible future line of work on Section 8.2.

6.3.3 EQUIVALENCE statements

Description

An EQUIVALENCE statement in FORTRAN is used to specify that a set of entities are to share the same memory location. It has the form:

\[
\text{EQUIVALENCE}(a_1, a_2, \ldots, a_m), (b_1, b_2, \ldots, b_n), \ldots
\]

where each set between brackets is a list of entities that are to share the same storage unit, and each of those entities is either a variable name, array element name, array name, or character substring name.

For example, the following EQUIVALENCE statement:

\[
\text{EQUIVALENCE}(I, J, K), (X, Y, A, B), (Z_1, Z_2)
\]

where the variables I, J and K are of type INTEGER, the variables X, Y, A and B are of type REAL, and the variables Z1 and Z2 are of type COMPLEX, would result in the memory allocation and mapping illustrated in Figure 6.3. This means that the names I, J and K may be used interchangeably to refer to the same memory location in the program unit in which the EQUIVALENCE
Figure 6.3: How entities share memory locations through \texttt{EQUIVALENCE} statements.

\begin{center}
\begin{tabular}{|c|c|}
  \hline
  I & J \leftrightarrow K \\
  \vdots & \vdots \\
  X & Y \leftrightarrow A \leftrightarrow B \\
  \vdots & \vdots \\
  Z_1 & Z_2 \\
  \hline
\end{tabular}
\end{center}

statement occurs, and similarly for the names \(X, Y, A\) and \(B\), and for the names \(Z_1\) and \(Z_2\).\footnote{Note that the FORTRAN 77 standard specifies that \texttt{COMPLEX} values, as well as \texttt{DOUBLE PRECISION} values, should occupy two numeric storage units in memory.}

\texttt{EQUIVALENCE} statements can easily become very complex, particularly when used with arrays or array elements. For example, consider a program containing the following statements:

\begin{verbatim}
INTEGER A(2,3),B(3),C(5),SUB
EQUIVALENCE(SUB,A(1,2),B(2),C(3))
\end{verbatim}

As FORTRAN 77 holds array elements in consecutive storage units, this \texttt{EQUIVALENCE} statement establishes an association not only between \texttt{SUB},
Figure 6.4: How array elements can become superimposed through the use of \texttt{EQUIVALENCE} statements.

\begin{verbatim}
A(1,1) <- C(1)
A(2,1) <- B(1) <- C(2)
A(1,2) <- B(2) <- C(3) <- SUB
A(2,2) <- B(3) <- C(4)
A(1,3) <- C(5)
A(2,3)
\end{verbatim}

$A(1,2)$, $B(2)$ and $C(3)$, but also among various other entities, as seen in Figure 6.4. Note that if an array name is used inside a list of entities, this is considered as being an array element name, corresponding to the first element of that array. This means that the statements:

\begin{verbatim}
REAL A(2,3),B(3),C(0:5)
EQUIVALENCE(A,B(2),C)
\end{verbatim}

have the exact same effect as the statements:

\begin{verbatim}
REAL A(2,3),B(3),C(0:5)
EQUIVALENCE(A(1,1),B(2),C(0))
\end{verbatim}
Problems and solution

As seen in the previous example, EQUIVALENCE statements can easily become very complex. There are seldom good reasons for using them, and devious bugs can be introduced (Balfour & Marwick, 1988) by using EQUIVALENCE statements in ways that are not completely straightforward. In particular, there are two ways in which EQUIVALENCE statements may become very difficult to understand, and with sometimes unpredictable behaviour:

- when mixing data types within a list in an EQUIVALENCE statement; although FORTRAN 77 permits this,\(^{10}\) these EQUIVALENCE statements will have a different behaviour depending on the underlying hardware;

- when using entities in EQUIVALENCE statements that reside in common blocks; although this is allowed by FORTRAN 77 with a few minor restrictions, the resulting behaviour can become extremely difficult to understand and analyse.

Like COMMON statements, the mapping of different names to the same memory locations is quite difficult to achieve with the current implementation of Papa, and would require the use of a very complex symbol table, complete with expression analysis and value memory for each variable. For this reason, and also to avoid the need to introduce hardware specific data, the following approach to deal with EQUIVALENCE statements has been adopted:

- if only single variables (i.e. not part of an array) share a memory location, then if one of those variables is used or modified, all variables sharing the same memory location are reported as being used or modified;

- if elements of an array are included in a list of entities sharing the same memory location, then whenever an element of that array is used or

\(^{10}\)The FORTRAN 77 standard only prohibits the equivalencing of entities of type CHARACTER with entities of other types.
modified, all variables or arrays sharing the same memory location as any of the elements of the array are reported as being used or modified.

Taking the example illustrated in Figure 6.4, this means that the following line of code:

$$A(1,1) = 10$$

will have the following DEF set:

$$DEF = \{ A(1,1), B, C, SUB \}$$

Note that although this again is a solution that somewhat limits the range of transformations that Paragen can evolve, in reality most array references are made to an undefined element (e.g. $A(I)$); not knowing which element of array $A$ has been accessed means that a worst case scenario must be assumed, and therefore all variables and array elements which might share a memory location with element $A(I)$ must be assumed to have been accessed, which in effect is what happens with the approach explained.

### 6.3.4 ENTRY statements

**Description**

The non-executable ENTRY statement is used in FORTRAN to specify an alternative entry point to a function or subroutine. It has the form:

```
ENTRY name (arg1, arg2, ..., argn)
```

where *name* is the symbolic name associated with this entry point, and $(arg1, arg2, ..., argn)$ is a list of dummy arguments separated by commas (the argument list may be empty, in which case the enclosing brackets are optional).
An ENTRY statement allows a procedure reference to begin at a specific executable statement, within the function or subroutine subprogram in which that ENTRY statement appears. It may appear anywhere within a function after the FUNCTION statement, or within a subroutine after the SUBROUTINE statement, except that an ENTRY statement must not appear between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO-loop.

The following is an example of the use of ENTRY statements:

```fortran
FUNCTION AREA(A,B,C)
  PARAMETER (PI=3.14,FAC=PI/180)
  S=0.5*(A+B+C)
  AREA=SQRT(S*(S-A)*(S-B)*(S-C))
  RETURN

ENTRY AREA1(A,ANGA,ANGB,ANGC)
  X=A*SIN(FAC*ANGB)/SIN(FAC*ANGA)
  AREA1=0.5*A*X*SIN(FAC*ANGC)
  RETURN

ENTRY AREA2(A,B,ANGC)
  AREA2=0.5*A*B*SIN(FAC*ANGC)
END
```

In another program unit, references to the symbolic names AREA, AREA1 or AREA2 will result in a different set of instructions being executed, as shown in Table 6.8.

The dummy arguments associated with different ENTRY statements need not agree in order, number, type or name, nor do they have to agree in order,
Table 6.8: How the specification of different entry points translates to different sets of instructions being executed.

<table>
<thead>
<tr>
<th>Calling statement</th>
<th>Statements executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA(P,Q,R)</td>
<td>S=0.5*(A+B+C)</td>
</tr>
<tr>
<td></td>
<td>AREA=SQR(S*(S-A)<em>(S-B)</em>(S-C))</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
</tr>
<tr>
<td>AREA1(P,Q,R,T)</td>
<td>X=A<em>SIN(FAC</em>ANGB)/SIN(FAC*ANGA)</td>
</tr>
<tr>
<td></td>
<td>AREA1=0.5<em>A</em>X<em>SIN(FAC</em>ANGC)</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
</tr>
<tr>
<td>AREA2(P,Q,R)</td>
<td>AREA2=0.5<em>A</em>B<em>SIN(FAC</em>ANGC)</td>
</tr>
</tbody>
</table>

number, type or name with the dummy arguments originally specified in the FUNCTION or SUBROUTINE statement of the program unit.

Problems and solution

The rules for the ENTRY statement are very complicated, and go against one of the main principles of structured programming, which is that each program unit should have exactly one point of entry; it is therefore generally better to use a set of separate program units to achieve the same result (Page, 1995).

In the case of Papa, the use of ENTRY statements can make its task very difficult, as different entry points can reference the same function or subroutine (through different names), but using a possibly different order, number, type and name for their arguments. Also, it is very difficult (and sometimes impossible) to determine exactly the set of instructions that are to be executed when a call to a specific entry point is made.\textsuperscript{11}

For these reasons, the solution found was to treat each ENTRY statement as being a separate program unit. The following approach is therefore taken:

\textsuperscript{11}GO TO statements, or IF statements driven by user input, could easily disrupt the instruction sequencing of the program unit, making it impossible to associate specific sets of instructions to entry points.
• for each program unit parsed, there will be a different data file created for each of its entry points, but all statements from that program unit will be considered, when extracting information about used and modified arguments (since it might be impossible to know exactly which statements will be executed for each entry point);

• whenever a call to an entry point is found on a program unit, the data file corresponding to the symbol name specified in the referring statement is accessed, in exactly the same way as a data file for a normal function or subroutine call is accessed.

This approach is fairly robust; to improve it, the control flow of each program unit containing ENTRY points would have to be analysed, and only if a specific set of instructions could be associated with each entry point, could more accurate data be saved onto the data file related to each entry point.

6.3.5 Input and output statements

Description

Like any other programming language, FORTRAN 77 contains a set of statements that control the exchange of data between the program and external devices, be it the screen, keyboard, card readers, line printers, magnetic tapes and disks, timesharing terminals, etc. These are:

READ statements : also called input statements, these are used to read data from a file, or from the designated input unit (usually the keyboard); a set of options known as input/output control specifiers can be used to specify which unit the data will be read from, the format and length of the data, etc.

WRITE and PRINT statements : also called output statements, these are used to write data to records in a file, or to the designated output unit (usually the screen); as with READ statements, WRITE statements can be used with input/output control specifiers (the PRINT statement
having the same behaviour as simple WRITE statement writing to the designated output unit).

**OPEN statements**: these can be used to connect an existing file to a unit, to create a new file and connect it to a unit, or to change certain options of the connection between a file and a unit; a set of options known as *connection specifiers* control which unit to use, the name of the physical file, the access type, etc.

**CLOSE statements**: these are used to disconnect a file from a unit; a set of options known as *close specifiers* is used to control which unit to disconnect, as well as to collect specific information about the close process (e.g., success of the operation).

**INQUIRE statements**: these statements may be used to investigate properties of a file or a unit; a set of options known as *inquire specifiers* is used to specify which unit or file to use as a source, and to collect various kinds of information (such as type of access, format and length of records, etc.)

**END FILE statements**: these, when executed, cause an end-file record to be written to the sequential access file connected to the specified unit; it can be used with a list of options known as *position specifiers*, which specifies the unit to work on, and collects information about the success or otherwise of the operation.

**BACKSPACE statements**: the execution of these statements causes the file connected to the specified unit to be positioned before the preceding record (if the file is at its initial point, the statement has no effect); as with **END FILE** statements, a *position specifier* list may be used as an argument.

**REWIND statements**: these, when executed, cause the file connected to the specified unit to be positioned at its initial point (if the file is
at its initial point, the statement has no effect); as with the previous two types of statements, a position specifier list may be used as an argument.

**Problems and solution**

If a unit is accessed (either to use its records or to modify them), no two accesses to that unit can be made in parallel; therefore, Papa needs to introduce a dependency between accesses to files connected to units. The way to do this is to use the unit specifier connected to each file as a modified variable, whenever that file is accessed.\(^{12}\) If, for example, an OPEN statement connects unit 57 to a file, a WRITE statement writes some data using the same unit, and finally a CLOSE statement disconnects unit 57 from the file it was originally connected to, all these statements will be reported as modifying variable 57; this will introduce a data dependency between the three statements, preventing Paragen from evolving transformations that will execute any of those statements in parallel (or in a different order).

There are two special cases that need to be taken into consideration when using this technique, however:

- when the unit accessed is the designated input or output unit;
- when the unit specifier is a variable.

In the first case, the unit specifier is the symbol * (READ, WRITE and PRINT statements may be used without a unit specifier, which means they are used to access the designated input or output unit); therefore, whenever a statement is used to access the designated input or output units, the symbol * will be reported as being a modified variable.

In the second case, it is impossible to tell which unit has been accessed, and therefore which file has been accessed (unless the value of each variable is known at any point in the program execution), so whenever a statement

\(^{12}\text{Hence the appending of a letter 'f' when using GO TO statements (Section 6.2.2), so as not to confuse those with unit specifiers.}\)
accesses a file that has a variable as its unit specifier, this statement cannot be executed in parallel with any other statement accessing a file. If such a statement exists in the code being parsed by Papa, then a special variable I/O will be declared as being modified on any input or output statement, except for statements accessing directly the designated input or output units (as the \(/\) symbol is not a valid symbol in a FORTRAN 77 variable name, there is no danger of the newly introduced variable being confused with an existing variable name in the code).

6.4 Conclusion

As a pre-processor to the language-independent Paragen process, Papa has to extract information from the original source code, and present it to Paragen in exactly the same way as a pre-processor for a different programming language would present it. This means that all the specific features of the underlying language must be abstracted from the information to be presented. Each programming language has its own specific problems; being a language firstly introduced in the 1950s, and a standard introduced in the 1970s, the FORTRAN 77 language has many features that do not appear in today’s modern languages. These need to be masked when extracting parallelising information, in a way that will prevent Paragen from evolving transformations that will modify the semantic meaning of the original code.
Chapter 7

Implementation

The previous chapters have looked at the background of this work, and moved on to specifying the objectives that Papa should achieve. In this chapter, the development of the tool is described. As with any software project, certain choices had to be made, such as the development tools to use, and decisions about the flow and transformation of information, from the time it is accepted as input until it is produced as output.

7.1 Development choices

7.1.1 Tools

When developing a tool that will parse formatted text as input, there is a wide choice as to what development tools to use. Certain scripting languages, such as Perl (Wall et al., 1996) or Awk (Aho et al., 1979; Aho et al., 1988), provide convenient text handling features, but leave it to the programmer to specify exactly what to look for.

Since the objective of Papa is to parse a program that follows the FORTRAN 77 standard, a tool that accepts a grammar representing its syntax had to be chosen. The choice fell on the use of the lex and yacc tools (Levine et al., 1995), as they are tools specifically designed to write compilers and interpreters, and are widely accepted and supported on most systems.
Lex and yacc were both developed at Bell Laboratories in the 1970s. Yacc was the first of the two, developed by Stephen C. Johnson (Johnson, 1975), whereas lex was designed by Mike Lesk and Eric Schmidt (Lesk & Schmidt, 1975) to work with yacc. Both tools have been part of the standard set of UNIX tools since 7th Edition UNIX.

The versions of these tools used for the development of Papa were the GNU project tools flex (“Fast Lexical Analyser Generator”), developed by Jef Poskanzer, Vern Paxson and Van Jacobson (Paxson, 1995), and bison, a yacc replacement written by by Robert Corbett and Richard Stallman (Donnelly & Stallman, 1991). They were used, however, with a compatibility flag, for backwards compatibility with the original AT&T versions, and will therefore be referred to as lex and yacc.

**Lex**

Lex is a tool for the design of tokenisers (or lexers, hence its name), designed to work with yacc. In the implementation of Papa, lex was used to develop a lexer that receives the text of the original FORTRAN 77 source code, and transforms it onto a sequence of tokens (or lexems); these are then sent to a parser developed with yacc. These tokens represent the entities that each statement is composed of. For example, in the following statement:

\[ X = A \times 3.14 \]

the lexer will analyse the characters that the sentence is made up of as it reads it from the source file, and will send a tokenised sequence to the parser:

\[ \text{NAME}(X) \ \text{EQUAL} \ \text{NAME}(A) \ \text{TIMES} \ \text{RCONST}(3.14) \]

Notice how certain tokens have values associated with them, such as names for variables, or the value of the real constant 3.14; these values will be required by the parser for further analysis.
Yacc

The yacc ("yet another compiler compiler") program is used to create parsers that receive a sequence of tokens from a lexer, and then parse that sequence according to a BNF (Backus-Naur Form)\(^1\) grammar. In the case of Papa, a BNF grammar for the FORTRAN 77 language was used with yacc to generate a parser; this parser can therefore be used to recognise the syntax of a FORTRAN 77 statement, and execute commands when such recognition occurs.

A BNF grammar (Backus et al., 1960) is a formal way of representing the syntax of a context-free or context-sensitive language (Chomsky, 1956), through a collection of production rules. BNF grammars make use of two kinds of symbols: *terminal* symbols, which represent symbols that appear in the language, and *non-terminal* symbols, which are symbols that can be expanded into one or more terminal and non-terminal symbols, and which are used on the parsing and recognition phase.

Using the previous example, the production rules from the FORTRAN 77 grammar (or rather, simplified versions of those) that will recognise the sequence of tokens, sent by the lexer, as an assignment statement, would be the following:

\[
\begin{align*}
(0) \text{assignment Stmt} & : \text{var EQUAL expr} \\
(1) \text{var} & : \text{NAME} \\
(2) \text{op} & : \text{PLUS} \\
(3) & \mid \text{MINUS} \\
(4) & \mid \text{TIMES} \\
(5) & \mid \text{DIVIDE} \\
(6) \text{expr} & : \text{var} \\
(7) & \mid \text{RCONST} \\
(8) & \mid \text{expr op var} \\
(9) & \mid \text{expr op RCONST}
\end{align*}
\]

---

\(^1\)And not a “Backus Normal Form”, as it is often mistakenly referred to (Knuth, 1964).
Note that the grammar notation shown is the one used internally by yacc (not the original Backus notation), and that the convention of using capitalised words for terminal symbols and lowercase words for non-terminal symbols has been adopted.

The recognition phase occurs as follows: the parser will receive the first token (`NAME`), so its read buffer will look like this:

```
NAME
```

Since there is only one rule (rule 1) that recognises this terminal symbol, the token `NAME` is transformed\(^2\) into the non-terminal symbol `var`:

```
var
```

Next, the following token from the lexer is read onto the buffer:

```
var EQUAL
```

Since there is no rule that recognises the sequence (or part of the sequence) currently in the buffer, no action is taken, and therefore the next token is read onto the buffer:

```
var EQUAL NAME
```

This process continues until the whole sequence of tokens has been processed; Figure 7.1 shows the parse tree for the example expression.

The extraction of used and modified variables follows closely this recognition phase; indeed, when a token has been identified as a `NAME` by the lexer, no decision can be made about whether this is a used or modified variable; it is only through the parsing process that the context in which that symbol

\(^2\)The term *reduced* is sometimes employed.
Figure 7.1: Resulting parse tree for the example sequence of tokens; circles represent non-terminal symbols, and rectangles represent terminal symbols.

\[
\text{assignment\_stmt}
\]

\[
\text{var} \quad \text{EQUAL} \quad \text{expr}
\]

\[
\text{Name} \quad \text{expr} \quad \text{op} \quad \text{RCONST}
\]

\[
\text{var} \quad \text{TIMES}
\]

\[
\text{Name}
\]

appears can be established, and therefore the decision of reporting it as being a used or modified variable can be taken.

For example, in the grammar shown, the non-terminal symbol \text{var} in rule 0 should be reported as a modified variable, whereas all \text{var} symbols used to build the non-terminal symbol \text{expr} in that same rule should be reported as being used variables.

7.1.2 Grammar

As Section 4.3.1 explained, the use of a standard FORTRAN 77 grammar on its own would be problematic, as there are many vendor specific extensions that were added after the standard was published, and that are widely accepted and used. To be compatible with as much legacy code as possible,
many of those extensions were used to extend the FORTRAN 77 grammar used with yacc to build a parser. The following is a list of the extensions that are accepted:

**Use of lower case letters**: these will be interpreted as capital letters, except inside character strings (which are never modified) and inside comment lines (which are ignored by Papa).

**Use of double quotes**: they are interpreted as single quotes.

**Use of TAB character**: it is interpreted as 6 spaces if found at the beginning of a line, or as a single space anywhere else, except inside text constants.

**Use of _ and @ in variable names**: these characters are treated the same way as alphabetic characters.

**Use of long variable names**: the standard specifies 6 as the maximum length of a variable name; in Papa, any length up to a user-defined maximum is accepted.

**Use of characters other than C and * to start comment lines**: lines with the characters c, C, *, or ! in the first column are treated as comments.

**Use of in-line comments**: if the character ! is used outside a text string, the remainder of that line is treated as a comment.

**Lines longer than 80 characters are accepted**: according to the standard, all text from columns 73 through 80 is ignored, and no line may be longer than 80 columns;³ in Papa, lines of any length are accepted, and all characters from column 73 on are simply ignored.

³This specification is clearly for historical reasons: the IBM 704 was programmed by means of punch cards, and could only read the first 72 characters of an 80 column card. Furthermore, on a punched card, the card sequence number is often punched in columns 73 to 80, and was ignored by the early FORTRAN compilers.
Use of any characters inside character strings: accepted.

Use of the non-standard `DO...END DO` statement: accepted.

Use of the non-standard `TYPE` statement: accepted.

Statements with any number of continuation lines: the standard allows a maximum of 19.

Use of the `IMPLICIT NONE` statement: the meaning of this statement is that all variables must have their data types explicitly declared. Papa will accept its use, but will not verify its correct use (see Section 4.3.2).

Use of optional precision specification for datatypes: the datatypes `INTEGER`, `REAL`, `COMPLEX` and `LOGICAL` can be followed by an optional precision specification; for instance, `REAL*8` means an 8-byte floating point data type.\(^4\)

### 7.2 Parsing the code

In this section, the actual process of parsing the code is explained. The technique used was to make three passes through the source code: a first pass to format it, a second pass to extract required semantic information, and a third pass to extract the parallelising information to produce as output. The function of each of these passes is explained in detail in the next sections.

#### 7.2.1 First pass

The objective of the first pass is to accept the original source code as input, and produce a new functionally equivalent code as output, which will then be parsed by the second and third passes. As only the format of the code will be analysed, regardless of the statements used, the parser will not be used

\(^4\)REAL*8 is an alternative to DOUBLE PRECISION, introduced by IBM and used also by Digital; several variants exist.
in this first pass; only the lexer will accept the characters from the original source code, and will produce a new functionally equivalent code, with the following changes:

**Broken lines are joined**: because of the limited line length of FORTRAN 77, a mechanism known as continuation lines was introduced; continuation lines, as their name suggests, continue the statement of the previous code line, and are identified by placing a single character on column 6. The existence of a statement broken over several lines goes against the way Paragen works, because Paragen’s decision of swapping statements or executing them in parallel is based on line numbers (for example, execute lines 34 and 36 in parallel); therefore *Papa* produces after its first pass a Fortran code with all broken lines joined, so that a line number corresponds to a single full statement.\(^5\)

**Characters from column 73 on are discarded**: in FORTRAN 77 programs, it is a common technique to place text after the 72\(^{nd}\) column; this text can be used as a comment about the function of the line. This comment text will not be copied onto the output of the first pass, as it is not part of the statement to be executed, and complicates the process of joining broken lines.

**Comment lines are removed**: a comment line in FORTRAN 77 is identified by placing a `C` or `*` character in the first column of the current line; they are useful if the legacy code is to be analysed by a person, but useless if the parallelising process is automatic. Comment lines can make the task of joining a broken line quite complex, specifically if located between a line and its continuation line(s); they are therefore removed from the output of the first pass.

\(^5\)The output of the first pass is therefore used as if it were the original source code when applying Paragen’s transformations (van Hemert, 2002); when the parallelised code will be compiled, however, lines longer than 72 characters will have to be broken again.
Empty lines are removed: empty lines help the analysis of the code by a person, as they make the code easier to read, but are of no use on an automatic process, and also render the process of joining broken lines more difficult; they are therefore removed from the output of this pass.

### 7.2.2 Second pass

The second pass parses the output of the first pass with the FORTRAN 77 grammar. This second pass is used to gather information about certain qualities of the code, and to build specific symbol tables that need to be complete before any information is to be passed to Paragen.

Some of the information gathered in this pass is needed for the application of the techniques described in Chapter 6. The core semantic information extracted in the second pass is therefore the following:

- list of arguments to the current program unit;
- list of arguments which are used as function pointers;
- list of arguments which are used as arrays;
- list of statement labels which are used in GO TO, ASSIGN, and arithmetic IF statements;
- list of which (if any) variables have been used as unit specifiers, in input or output statements.

Since this information is required from the whole source code, before the DEF and USE sets for each individual statement can be computed, it has to be extracted in this second pass. No information is therefore written onto the output file at this stage.

### 7.2.3 Third pass

After the required information about the whole code is extracted in the second pass, the task of writing the DEF and USE sets for each line of code
to an output file, which can then be passed to Paragen, can be accomplished in a final pass through the code.

The following is a list of the key data collected from the code in this pass, to allow the extraction of the information needed by Paragen (listed in Section 4.3.2):

- Type of each statement;
- list of declared arrays;
- list of variables referring to entities inside common blocks;
- variable names included in EQUIVALENCE sets;
- list of entry points.

This is information that is extracted as it appears in the code, and complements the information from the second pass, allowing for the correct extraction of DEF and USE sets from each statement.

### 7.2.4 Solving specific Fortran problems

As mentioned previously (Section 4.2.3), FORTRAN 77 was developed well before the separation of lexical analysis and syntax analysis was considered the best way for compiler design, and therefore contains certain features that make it difficult to parse the source code, and to extract the correct information from it.

Below are the main difficulties, and the way they were dealt with during the development of Papa.

**Backward context**

The absence of reserved keywords complicates the task of the lexer, which is to map expressions from the original code into tokens, which will be used by
the parser as terminal symbols through its grammar. For example, a statement such as:

\[ X = DO \]

which should be tokenised as:

\[ \text{NAME}(X) \text{ EQUAL } \text{NAME}(DO) \]

could be easily tokenised as:

\[ \text{NAME}(X) \text{ EQUAL } DO \]

that is, the variable called \textit{DO} might be mistakenly detected as being the keyword \textit{DO}, which would produce an error when the parser attempts to recognise this sequence of tokens.

To avoid incorrect detections like this, the lexer has to establish the backward context of each expression, before making the decision of which token to associate that expression with.

\textbf{Forward context}

Similarly, in certain cases it is necessary to look ahead at a certain number of characters that follow a symbol, to be able to correctly tokenise that symbol. For example, the following FORTRAN 77 statement:

\[ DO = X \]

which should be tokenised as:

\[ \text{NAME}(DO) \text{ EQUAL } \text{NAME}(X) \]
could be easily tokenised as:

```fortran
DO EQUAL NAME(X)
```

which is a sequence of tokens that does not correctly represent the valid structure of a `DO` statement, and that will not be recognised by the used FORTRAN 77 grammar.

To avoid this problem, the lexer code needs to look ahead at a certain number of characters before recognising a sequence of characters as a specific token, which goes against the principle of lexer/parser separation.

**Nesting levels**

Nesting levels are quite easy to calculate for `IF` (and `DO...END DO`) statements, but on certain classical `DO` statements they can be less so, as a single labelled statement (usually a `CONTINUE` statement) can be used to terminate several `DO` statements. For example, consider the following code, initialising an array of $N \times N$ elements to the value 0:

```fortran
DO 10 I=1,N
  DO 10 J=1,N
    A(I,J)=0
10 CONTINUE
```

If the nesting level of the first statement shown is level 0, then the nesting level of the second is 1, and the level of the third and fourth statements is 2. However, the next statement (following the statements shown) should have its nesting level set to 0, and not 1, as one might expect (the existence of a single `CONTINUE` statement could induce that error).

To properly calculate nesting levels after `CONTINUE` statements, **Papa** make use of a stack of statement labels; every time a `DO` statement makes reference to a label, that label is pushed onto the stack, and the nesting
level is incremented. When a labelled statement is reached, if its label is on the top of the stack then it is popped, and the current nesting level is decreased; this is done as long as the top of the stack matches the label of the current statement. This ensures that when a labelled statement is used to terminate several DO statements, the nesting level of the statement following it is correctly calculated.

7.3  Testing

There are two main objectives for Papa:

- parse FORTRAN 77 compliant code;
- correctly extract the information required.

To assess the scale to which these objectives were achieved, a set of test source files were used, and are described in this section, along with an example of one such test file.

7.3.1  Quantity testing

Quantity testing was used to ensure that the parser is able to recognise constructions seen on most FORTRAN 77 programs. In order to do so, batches of FORTRAN 77 programs were used with Papa, and all source code files that Papa was not able to parse would then be checked (to check for example if any FORTRAN 77 extensions were used), and if necessary the grammar used with Papa was updated.

The following is a list of some of the source code used on this quantity testing process:

The Livermore benchmarks (McMahon, 1986) : sometimes known as the Livermore loops, this is a collection of kernels that measure Fortran numerical computation rates for a spectrum of CPU-limited computational structures. They are often used for analysis of parallel code.
compilers (Bozkus et al., 1994) and instruction level parallelism (Smith et al., 1989; Davidson & Jinturkar, 1995), amongst other uses. They consist of 31 program units, with a total of over 5700 lines of code.

**The MDBNCH benchmark** (Ercolessi, 1988): the MDBNCH is a molecular dynamics benchmark; it can be used to provide information about the speed of machines on codes which are relatively small, but tend to access memory in an irregular way. The code is written in 100% ANSI-standard FORTRAN 77, to ensure maximum portability, and provides therefore an excellent test for the conformity of Papa. The code consists of 28 program units, with over 2000 lines of code.

**The Linpack benchmark** (Dongarra et al., 1979): the Linpack Benchmark is a measure of a computer’s floating-point rate of execution; it is determined by running a computer program that solves a dense system of linear equations. This benchmark is typically used to measure computer performance (Mattson & Henry, 1998), since computational linear algebra is at the heart of many scientific problems; the results for many architectures using this benchmark (Dongarra, 1989) tend to be used as standard references (Zhou et al., 1998; Caprara et al., 1998). This software is currently divided onto three benchmark computations, spread across 11 program units, with a total of around 800 lines of code.

**The g77 test suite** (GCC Steering Committee, 2002): this is a continuously growing collection of several small FORTRAN 77 programs, which are particularly difficult to parse and compile; they are used by the gcc development team to test their FORTRAN 77 compiler, g77. These tests are designed for compiler testing, and are therefore very challenging, providing an opportunity for testing Papa in extreme examples. They are composed of over 150 programs of various sizes, with over 9000 lines of code; from these, 40 were randomly chosen to test Papa.
The LAPACK library (Demmel & Kahan, 1990): this is a well known and widely used library, which makes use of block matrix algorithms, and is written for good data reuse on a machine with a two-level memory hierarchy (Anderson et al., 1995; Kodukula et al., 1997). The CBDSQR routine, which computes the singular value decomposition of a real $N \times N$ bidiagonal matrix was used; it consists of 750 lines of code.

Varied source code: these are publicly available FORTRAN 77 source codes from several web locations (Lahey Computer Systems, 2002; The Fortran Company, 2002; Fortran Library, 2002). They are mainly composed of scientific applications (such as calculations of air flow, autocorrelation tests for pseudo-random numbers, etc), and were used to broaden the range of different programs Papa was applied to. They consist of over 70 program units, with a total of over 5000 lines of code.

In its current state of development, Papa is able to parse all the listed code. Other benchmark suites are also available and widely used (Hockney & Berry, 1984; Berry et al., 1989; Standard Performance Evaluation Corporation (SPEC), 1989), if further testing is required.

7.3.2 Quality testing

Quality testing was used to ensure that Papa is able to properly extract the required information from source FORTRAN 77 programs. For each test program, the exact information which should be extracted from each line was pre-calculated by hand, and placed in a file (using the same format as Papa); an automated process would then compare the contents of these files with the output of Papa, for each of the test programs, to ensure that all the required information was properly extracted.\(^6\)

\(^6\)Papa was able to extract all the required information from the used test files; on most of the cases where a problem appeared to occur, it turned out that Papa was extracting the required information, while the human generated data was incomplete, which shows the complexity of the task, and the need to automate it.
There were two main types of source code used in this testing phase:

**The Papa test suite** : this is a collection of programs designed exclusively for use with Papa, to test its ability to deal with specific problems concerning the extraction of information, particularly those highlighted in Chapter 6; each program focuses on a specific problem, like GOTO statements, COMMON statements, etc. The suite is composed of over 25 program units, with a total of around 700 lines of code.

**The Livermore benchmarks** : the parallelising information from these was extracted by hand, and compared to the output of Papa, by the automated process described above.

### 7.3.3 Example

As an example of the test files used with Papa, consider the code shown in Table 7.1. This is a code that calculates the average of \( n \) numbers, but its variable names were chosen to specifically test the ability of Papa to extract the correct DEF and USE sets from a code using some very unconventional variable names, that could be easily mistaken as keywords.

The output generated by Papa, which is the correct output, is summarised in Tables 7.2 and 7.3. The correct extraction of this information involved mainly the ability of establishing a backward and forward context, for generating the correct sequence of tokens to send from the lexer to the parser.

Several traps exist in the code; notice the use of variables called PROGRAM, END, PRINT and IF, testing the ability of the lexer to correctly establish the context in which these variables appear. Of particular difficulty is the expression shown in line 6, where the use of the variables IF, LE and GT results in code which, although correct, is very hard to understand for a human reader; the correct tokens for that statement would be:

```plaintext
IF ( NAME(IF) LE NAME(LE) OR NAME(IF) GT NAME(GT) ) THEN
```
Table 7.1: Example test file from the Papa test suite; the code calculates the average of $n$ numbers, introduced by the user.

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROGRAM PROGRAM</td>
</tr>
<tr>
<td>2</td>
<td>INTEGER END,PRINT,LE,GT</td>
</tr>
<tr>
<td>3</td>
<td>PARAMETER(LE=0,GT=50)</td>
</tr>
</tbody>
</table>
| 4    | PRINT *,'How many numbers (max = ',GT,')?'
| 5    | READ *,IF |
| 6    | IF(IF.LE.LE.OR.IF.GT.GT) THEN |
| 7    | PRINT *,IF,' is not a valid number. Goodbye.' |
| 8    | STOP |
| 9    | ENDIF |
| 10   | PRINT=0 |
| 11   | PRINT *,'Type in the numbers, one at a time' |
| 12   | DO 10 END=1,IF |
| 13   | READ *,READ |
| 14   | PRINT=PRINT+READ |
| 15   | 10 CONTINUE |
| 16   | PRINT *,'Total =',PRINT |
| 17   | PRINT*,'Average =',PRINT/IF |
| 18   | END |
Table 7.2: Output produced by **Papa** for the test file (notice the inclusion of a separation line between lines 3 and 4, for the purposes of correct sequencing).

<table>
<thead>
<tr>
<th>Line</th>
<th>Nesting level</th>
<th>DEF</th>
<th>#</th>
<th>USE</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>{1st}</td>
<td>1</td>
<td>{}</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>{END,PRINT,LE,GT}</td>
<td>4</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>{LE,GT}</td>
<td>2</td>
<td>{1st,2nd}</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>{2nd,3rd}</td>
<td>2</td>
<td>{1st}</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>{*}</td>
<td>1</td>
<td>{GT,1st,3rd}</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>{IF,*}</td>
<td>2</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>{}</td>
<td>0</td>
<td>{1st,3rd,IF,LE,GT}</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>{*}</td>
<td>1</td>
<td>{IF,1st,3rd}</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>{1st}</td>
<td>1</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>{1st}</td>
<td>1</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>{PRINT}</td>
<td>1</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>{*}</td>
<td>1</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>{END}</td>
<td>1</td>
<td>{IF,1st,3rd}</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>{READ,*}</td>
<td>2</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>{PRINT}</td>
<td>1</td>
<td>{PRINT,READ,1st,3rd}</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>{1st}</td>
<td>1</td>
<td>{1st,3rd}</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>{*}</td>
<td>1</td>
<td>{PRINT,1st,3rd}</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>{*}</td>
<td>1</td>
<td>{PRINT,IF,1st,3rd}</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>{1st}</td>
<td>1</td>
<td>{}</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.3: Specific loop information extracted by **Papa** from the test file.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Nesting level</th>
<th>Iteration variable</th>
<th>Start value</th>
<th>End value</th>
<th>Increment value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-15</td>
<td>0</td>
<td>END</td>
<td>1</td>
<td>IF</td>
<td>1</td>
</tr>
</tbody>
</table>
7.4 Conclusion

In this chapter, specific information concerning the implementation of Papa has been provided. Papa uses a typical lexer/parser approach to extracting information from source code, but due to the specific characteristics of FORTRAN 77, certain adjustments had to be made to the classical approach. Also, the grammar which is used with the parser had to be extended, to accept some of the most used extensions to the FORTRAN 77 standard.

Three passes are then done through the source code: a first pass to format the code in a compatible way with Paragen’s transformations, a second pass to extract required information from the whole code, and a final pass to extract the specific information required by Paragen.

Finally, the issue of testing has been addressed, and an example test file was presented, illustrating the use of some of the techniques described.
Chapter 8

Conclusions

Software re-engineering is a field that has seen much work devoted to it, as a consequence of the enormous costs that the upkeep of legacy applications requires. The automation of this process is a promising field, as it aims to reduce those costs by a considerable amount; at the same time, the recent availability of low-cost multiprocessor hardware has refuelled the field of parallel computation, which in turn makes the auto-parallelisation of legacy code an effective way of not only taking advantage of more recent hardware, but also of considerably speeding up its execution time, all with little or no human interaction.

A project concerned with the concepts for serial to parallel code transformation is likely to be a complex one; the SCARE project, described in Chapter 3, is one such project, and Papa, the tool described in this thesis, is part of it. Its role is to extract specific information from legacy serial code; that information will then be used by another component of the project, Paragen, to convert that code into a semantically equivalent parallel program.

In this chapter, the research carried out for the development of Papa is summarised, and areas for future work are considered.
8.1 Summary

As a front-end to Paragen, a language-independent tool, Papa has to deliver the data required from the source program, in a way that does not need the knowledge of the specific structure and constructs of the programming language used, which Paragen does not possess.

The objectives accomplished by Papa can therefore be divided into two main areas: the extraction of the required information, and the encapsulation of the specific structure of the underlying programming language on that information.

Extraction of parallelising information

Paragen makes the decision about which transformations should be applied, by analysing the data dependencies which exist between statements; this means that Papa has to provide that information, plus other data (such as nesting levels and specific loop information) that will be required for the decision about which instructions can be swapped, and which can be executed in parallel.

In the simplest of cases, the use of a lexer and a parser is enough; through the use of a grammar, the context in which a variable name appears within a statement can be established, and the decision of whether that variable belongs to the DEF or USE sets can be made.

In FORTRAN 77, however, as in most other programming languages, the information required for each statement is often not self-contained; that is, the analysis of a single statement is not enough to provide all the data dependencies related to that statement. There are several cases where the analysis of further code is required: a function call, for example, requires the analysis of all the code of the referred function, to know which of the parameters passed to the function are used or modified, and which global variables are used or modified (from the use of COMMON statements); the use of EQUIVALENCE statements will also modify the USE and DEF sets, as
the reference to a single variable should be expanded to refer to all variables sharing its memory location;\(^1\) finally, due to the confusing syntax used by FORTRAN 77, the distinction between a function call (or a statement function declaration) and a reference to an element of an array depends on previous code, where the array may or may not have been declared.

These and other problems are solved through the use of symbol tables, explained in Chapters 6 and 7, which are built as the code is parsed.

**Language specific requirements**

Each programming language has its own specific characteristics. The FORTRAN 77 language, having been developed over 25 years ago (and compatible with even older standards), contains many facilities that are nowadays regarded as being *undesirable*: some of these are relics from earlier versions of FORTRAN (such as arithmetic IF statements), others contradict the principles of structured programming (such as GO TO statements, or alternate ENTRY points). On the other hand, many existing FORTRAN programs contain instances of these facilities, and analysing them is therefore required, so as to ensure the validity of the information to pass to Paragen.

As shown in Chapter 6, the usage of these kinds of statement greatly reduces the degree to which an existing serial program can be parallelised. That is to say, the quality of the input will determine the quality of the output, or at least will greatly influence it: if a program does not follow the principles of structured programming, then that program is unlikely to be transformed into a significantly faster parallel program, without changing its semantic meaning. For example, the program shown in Table 4.1 (Chapter 4) can be considered a good quality input, as it provides various opportunities for parallelisation, whereas the program shown in Table 6.4 (Chapter 6) is an example of a program that is unlikely to be transformed into a faster parallel program.

---

\(^1\)Much like pointers and unions in the C language.
Paragen chooses transformations for the original serial code based solely on data dependencies; this means that no consideration is given to the control flow inside that code. To ensure that control flow is not modified, Papa has to provide information about it, in a way which is transparent to Paragen, so as to maintain the language independence of that information.

As the only information passed to Paragen is sets of used and modified variables (plus nesting levels and specific loop information), all other information has to be encapsulated within this information. In the case of Papa, this was done by deliberately introducing data dependencies which did not exist in the original code; for this purpose, a set of variables was created (detailed in Section 6.2), and the dependencies created by those variables ensures that the control flow of the original code is maintained. Naturally, to ensure the control flow is not changed, the code needs to be analysed as a whole, before the information for each specific statement can be extracted; this is achieved by the use of a three-pass method (explained in Section 7.2).

8.2 Future work

The objectives set for Papa have been achieved; it is a fully developed software tool, which correctly extracts parallelising information from legacy code in the form of data dependencies, which ensure any transformations applied will keep the original semantic meaning of the code. Many extensions can be pursued from this work; the following is a list of some of them.

Expression simplification: as the atomic unit of Paragen is a statement, this means that each statement will be executed on a single processor, regardless of its complexity. Papa can be used to optimise this process: as it uses a lexer and a parser, it works from a fully decomposed expression, all the way to the recognition of a specific statement; in other words, it handles all the components of an expression, from its simplest form (a variable or a value) right up to to the complete expression (such as a complex mathematical expression). This therefore gives
Papa the ability of breaking down a complex expression into several less complex expressions, which can be executed in parallel. This process would involve the usage of a profiler to measure the complexity of each statement, and the calculation of dependencies by Papa, to ensure the expression can be broken down.

**Code inlining** : one of the principles of structured programming is that of modularisation, in which it is stated that the use of procedures greatly improves the readability and logic of a program, and facilitates later maintenance work; however, routine calls can be a relatively expensive operation, especially if the routine is small, due to the overhead work of the calling process. Code inlining merges the callee into the caller, removing the call overhead; it is particularly useful when a procedure is referred by a small number of statements, or when its code is too small to justify the call overhead. In the specific case of Papa, statement functions could easily be incorporated into their referring statements, due to their small size (they are composed of a single statement).

**Standard formatting of the information collected** : since all the information collected by Papa is to be used by Paragen, the format of this information is the one internally used by this tool. This information could however be useful for many other projects, rather than the SCARE project alone. For this purpose, a standard format could be adopted for the information extracted: a good choice would be the use of the Extensible Markup Language (XML), designed by the W3C (World Wide Web Consortium), which is a simple, very flexible text format derived from SGML (ISO Publications, 1986), as it provides an ideal format for storing structured data, intended for publishing or exchange between different applications.

**Aim at specific hardware and software** : some of the techniques employed, particularly those listed in Chapter 6, offer a conservative approach to most of the problems associated with FORTRAN 77; to
improve those solutions (and therefore allow for a higher degree of parallelism for the program to be generated), the utopia of designing a parallel information extraction tool for standard FORTRAN 77 would have to be abandoned, and efforts should concentrate on specific software suites, designed for specific hardware, and using specific compilers.
References


115


Index

Awk, 88

Compiler, 14, 35, 44, 46

Data dependencies, 12

FORTRAN
   analysis, 35
   arrays, 71
   history, 32
   i/o statements, 84
   instructions
      ASSIGN, 66, 96
      BACKSPACE, 85
      BLOCK DATA, 56
      CALL, 52, 55, 72
      CHARACTER, 80
      CLOSE, 85
      COMMON, 73, 108
      COMPLEX, 67, 78, 94
      CONTINUE, 65, 69, 99
      DATA, 56
      DOUBLE PRECISION, 78, 94
      DO loop, 37, 38, 68, 82, 94, 99
      ELSE, 69
      END DO, 94, 99
      END FILE, 85
      END IF, 69, 82
      END, 56, 103
      ENTRY, 56, 57, 69, 81, 109
      EQUIVALENCE, 77, 97, 108
      FORMAT, 56, 57
      FUNCTION, 52, 56, 82
      GO TO, 60, 96
      IF, 38, 67, 82, 99, 103, 109
      IMPLICIT, 56
      INQUIRE, 85
      INTEGER, 73, 94
      LOGICAL, 94
      OPEN, 85
      PARAMETER, 56
      PAUSE, 69
      PRINT, 84, 103
      PROGRAM, 56, 103
      READ, 84
      REAL, 94
      RETURN, 69
      REWIND, 85
      SAVE, 75
      STOP, 69
      SUBROUTINE, 52, 56, 82
      WRITE, 84
      statement sequencing, 55
Genetic Programming, 18
Grammar, 14, 37, 49, 90, 92
lex, 89
Lexical analysis, 14, 35, 97
MIMD, 9
MISD, 9
MPI, 8, 18
Paragen, 18
Parallel
    auto-parallelisation, 7, 11
    computing, 8
    information, 12, 38, 48
Parser, 13
Parsing, 13, 90, 94
Perl, 88
Pre-Processor, 26, 30
Programming paradigms, 13, 31
PVM, 8, 18
S.C.A.R.E., 2, 16, 29, 49, 107, 111
Semantic analysis, 35
SIMD, 9
SISD, 9
SoftDraw, 16
Softplan, 18
Software
    legacy, 5, 11
    maintenance, 6
    re-engineering, 5
SPMD, 9, 47

Syntax analysis, 15, 35, 97
von Neumann, 8, 9
XML, 111
yacc, 90