SYNTAX ORIENTED ANALYSIS OF THE RUN TIME PERFORMANCE OF PROGRAMS

by

Luis Felipe Cabrera

Memorandum No. UCB/ERL M81/30

13 May 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Syntax Oriented Analysis of the Run Time Performance of Programs

By
Luis Felipe Cabrera
Grad. (Catholic University of Chile) 1973
M.S. (Catholic University of Chile) 1975
M.A. (University of California) 1979
C.Phil. (University of California) 1978

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Applied Mathematics

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: [illegible] .... May 13, 1981
Chairman Date
Richard J Fateman .... May 13, 1981
J.W. Addison .... 13 MAY 81

....

Syntax Oriented Analysis of the Run Time Performance of Programs

by

Luis Felipe Cabrera

Syntax Oriented Analysis of the Run Time Performance of Programs

Luis Felipe Cabrera

ABSTRACT

In this thesis we consider the problem of finding efficient ways to determine, given the values for the input variables, the values of various performance indices associated with a program. For most purposes, we may concentrate on reproducing efficiently the dynamic profile of the program, i.e., on obtaining the exact profile for any run of the program as a function of the values of the input variables. Using the profile together with a data base of program performance information enables us to find the actual values of most of the desired performance indices.

To achieve this, we describe several kinds of *performance representations* of programs which express the profile equations of the program we are analyzing. We show that it is often possible to represent the profile equations by *program performance formulas*, whose evaluation time is linear in the length of their expressions. This is easily seen to be the best one can hope to obtain. We also delimit the cases in which an optimal performance representation can be found, and propose some alternative methods for the other cases.

In fact, our *skeleton* procedure can always be used, in the case of sequential programs, to represent the profile equations of a program. It is seen that, for compute bound sequential programs, the running time of the skeleton can be substantially shorter than that of running an instrumented version of the original program. Nevertheless, we also present examples which show that the running time of the skeleton need not be linear in the length of its text and in some cases is very close to the running time of the actual program.

A variety of solutions, and theoretical results which guide their usage, are presented to overcome the different problems due to the possible slowness of the skeleton, on the one hand, and to the non-applicability of the program performance formulae, on the other hand. It is recognized that most of the *definability* problems are caused by the iterations (loops). In particular, alternations (conditional statements) within iterations often cause a program not to exhibit an optimal performance representation. We examine in full detail the case when the actions on the control variables of an iteration are linear functions. In this case, we see that at run time we are even able to determine the exact pattern of truth values that a predicate has.

We have yet to implement a system which can build performance representations for us. However, we have discovered that known techniques used in data flow analysis suffice to obtain all the information we need about the variables in a program. Performance representations could be built by an "intelligent programming environment" while a program is being edited.

Our methods can also be used to obtain traces of programs efficiently. In fact, with minor modifications, the skeleton approach can be used to generate instruction traces of programs. A further refinement yields data traces. Moreover, those of our techniques which find faster performance representations can also be utilized to generate *condensed* traces, which can then be analyzed using a postprocessor. Once the *trace representation* of a program is built, obtaining several traces (corresponding to different sets of input data values) should be much more economical than actually running an appropriately instrumented version of the program several times.

When analyzing parallel programs, the problem of determining performance indices is much more complex since modeling the computational environment of the program is much harder. However, we show how our techniques for sequential programs can be used to aid this study.

[illegible]
Professor Domenico Ferrari
Computer Science Division
Chairman of Thesis Committee

TABLE OF CONTENTS

To Marcelle and Valentina

Acknowledgements

I am deeply indebted to Professor Domenico Ferrari, my thesis advisor. More than just introducing me to this area of Computer Science, he also provided an environment in which working was a pleasure, both here at Berkeley as well as in Pavia, Italy. I thank him specially for patiently and thoroughly reading and suggesting improvements to this thesis.

I thank Professor Richard Fateman and Professor John W. Addison, for valuable discussions concerning the research and for serving on my thesis committee.

I thank the members of the PROGRES group for their patience and willingness to discuss and suggest ideas. In particular, Ozalp Babaoglu, Newton Faller, Jehan-Francois Paris and Juan Porcar have provided me with numerous insights. Together we shared life, Fondue Fred, Hotel Basque and SOGESTA, many moments of unforgettable comradeship.

The most memorable efforts are joint efforts. Throughout these years as a student, my wife Marcelle has excelled in making my studies our project. Her attitude and support have been invaluable. Last but not least, the enthusiasm and cheerfulness of our young daughter Valentina have been a guiding force in these last months of intense work.

Financial Sponsors

I gratefully acknowledge the financial assistance received throughout my studies from the following sources:

Marcelle Stagno de Cabrera

Domenico Ferrari's NSF Grant MCS78-24618

Domenico Ferrari's NSF Grant MCS80-12900

Fulbright Foundation Fellowship 33711/15500168

Organization of American States Fellowship PRA-49330

Mario Italiani's Italian National Research Council Grant CNR-78.02548.07

CHAPTER 1

Introduction

The performance of software systems is becoming a central issue in the implementation and utilization of novel ideas and techniques in various fields of computer science. With hardware costs constantly decreasing, the availability of systems with a relatively large amount of main memory has become more widespread. However, these systems are now normally operated in a multiprogramming mode. Thus, they are very frequently utilized by many users at a time, creating contention for the installation's resources. For each of n users, service in a multiprogramming environment is often substantially inferior than it would be in a uniprogramming system of comparable (1/n) power. In fact, the behavior of any program in a multiprogramming system differs from that of the same program in a uniprogramming one because of the effects of the actions of other users as well as of the operating system. Programs almost always exhibit performance degradation, in terms of turnaround time, in multiprogramming systems.

It is not surprising then that the efficiency of software which is highly utilized (including in this category the operating system, which controls the activity of an installation) is becoming the object of an increasing number of performance evaluation studies. The purpose of these studies is to assess, and determine ways to increase, the productivity of an installation as well as to decrease the cost of processing certain important applications.

It has long been recognized that it is not enough to have very fast cpu's in order to complete a task in a predetermined amount of time, or to have an extremely efficient I/O subsystem: a system's performance depends on all of the aspects of its hardware and software configuration and on its workload. It is thus necessary to have software which uses appropriate algorithms and, most important, that makes suitable usage of the resources available in the installation. Unfortunately, today there are no software design tools, or methodologies, which allow us to analyze a symbiosis of this kind between a program and the installation in which it will run. Not enough work has been done in this area, although there is interest and need. In [Smi79, Smi80, Boo80], for example, we see discussions, implementations of methodologies and proposals on some of the issues that need be resolved to find a solution for this problem.

In the case of an existing program, when trying to analyze and/or predict its performance in a given installation, it is necessary to be able to determine exactly what resources and in what proportions the program requires to run. It would be very convenient if one could obtain this information in an efficient way, i.e., faster than by actually running the program and measuring it. We would like to have a *performance description* of the behavior of the program as a function of the values of its input variables, which would allow us to obtain efficiently the desired performance information. If such performance descriptions for programs were available, problems like comparing distinct software packages, or distinct implementations of a given algorithm, would become easier and less resource and time consuming.

Indeed, one can also envision that an intelligent operating system could use this performance description in its scheduling decisions, and thus allow for a better utilization of the available resources. Global optimization of

resource allocation becomes feasible if the resource demands can be predicted with reasonable accuracy. When a program is to run, the operating system is presented with the program and the input data; if it also were to get a performance description of the program of the type described above, allocations of resources could be made in a less uncertain situation, hence with a higher probability of success.

These representations could also aid us substantially in the study of the performance characteristics of a given program. Substantial execution time savings can occur for large numerical programs. Moreover, they could also be used to generate *traces* efficiently, and to assess program restructuring algorithms. Another application is in the detection of "dead code", i.e., code which is never executed.

1.1. Our Fundamental Problem

The *behavior* of a program means different things to people with different objectives. For example, one may be interested in the I/O activity, in the cpu requirements, in the number and type of arithmetic operations performed, in the amount of paging activity generated (in the context of a paged virtual memory system) or in the total running time. Each of these performance aspects of the execution of a program is normally a function of the value of the inputs to the program. It then becomes clear that the objective of the performance study must be established beforehand. Nevertheless, there exists a performance index which enables us to unify most of these studies. This index is a count of what gets executed in a run of a program.

A *basic block* is a linear sequence of program statements having one entry point (the first statement executed) and one exit point (the last statement executed). The *program profile* is a vector whose elements express the number of times each basic block is executed in a given run [Knu71b]. We shall often use the term *profile* to mean program profile. Thus, our profiles will be counts of basic block executions in a given run of the program.

Given a profile, it is rather simple to obtain several of the above mentioned performance aspects. The only one that may not be obtainable, depending on how intricate the flow of control structure is, is the dynamics of the memory demands produced by the program. As for all the other performance aspects, all that is necessary is some information which normally needs to be gathered only once per program-installation combination. In fact, what is required is just a map from the names associated with the basic blocks to the basic blocks of instructions, and then different maps from the basic blocks into tables of specialized information which occasionally may be installation dependent, although it does not necessarily have to be so.

For example, if we are interested in counting the different kinds of atomic operations that the program performs, then the map we need is one that associates with each basic block an itemized description of all the atomic operations performed by the statements in the basic block. Then, once we obtain the profile for basic blocks, we only have to multiply the value associated with a specific basic block by the number of times each atomic operation is performed in that basic block to obtain the counts of the operations executed. This procedure is certainly installation independent, but, once the necessary maps have been constructed for a program, they need never be recomputed.

However, if we are interested in estimating the running time of a program when executed with a given set of inputs, we need installation dependent information. In particular, if we assume a uniprogramming environment, what we need to find out is the (average) time each atomic operation takes as well as the execution time of each kind of branching statement appearing in the program.

For machine and assembly language programs this information may be obtained from tables supplied by the hardware manufacturer. If we are dealing with a program written in a high level language, then we need information which is both compiler and system dependent, because what we are really interested in is the (average) time a compiled atomic operation takes. This can be achieved by classical benchmarking techniques which call for performing the same operation a large number of times and then determining the (average) unit time. It should be clear that each time a new compiler is installed or a new system considered, a new table has to be obtained for the atomic operations we are interested in. Thus, estimating the running time of a program in different installations requires a different table for each installation.

A whole spectrum of subtler problems appear when we also want to consider the *effect* that an optimizing compiler for a high level language may have on the running time of a program. Our tables should now include context dependent information for the different kinds of code optimizations used by the compiler. In this case it is not a straightforward matter to determine the "cost" of an atomic operation. Conventional benchmarking techniques need to be applied carefully.

We shall call *profile equations* of a program those expressions which express the frequency counts of basic blocks as functions of the input data. Thus, if we had an appropriate representation for the profile equations, we would be able to obtain the profile of the program in an efficient way. The best achievable is to have an evaluation cost linear in the length of the representation of the profile equations.

The purpose of our study is precisely to explore different alternatives which will enable us to obtain profiles for programs in an efficient manner. In fact, we will describe automatic ways of representing the profile equations for a program and conditions under which they will yield the profiles with linear time evaluation cost. These methods will allow us to obtain profiles much faster than by actually running (a properly instrumented version of) the programs.

In Chapter 2 we introduce a "Program Performance Language". We define with it some syntactic objects which will represent (parts of) programs and others which will describe the values of variables at different points of a program. This will enable us to find the limits to our goals (within that formalization) as imposed by the topological complexity of the D-charts associated with programs, by the algebraic complexity of the modifications done to certain variables in a basic block, by the algebraic complexity of the predicate which governs a loop and by the interrelationships existing between distinct parts of the programs.

1.2. The Default Procedure

When we are interested in a count of the basic blocks as a function of the values of the input variables of a program and all we have is the program to work with, we will normally find that the program has many statements which do not play any role in this process. We shall now make this observation more precise.

Given a variable name x, we shall say that x is a *control variable* if its value affects the flow of control of the program. It is clear, then, that all

statements in the program which do not modify control variables may be excluded from an analysis whose sole purpose is to find the profile of a program.

We have thus found a first approach to our problem of efficiently generating program profiles. Given a program P we construct a program P_1 and suitable tables $T_1, \dots, T_n$, which will, we hope, enable us to obtain performance information about P much more rapidly than by actually running an instrumented version of P. This can be done in the following way:

(1) P_1 is obtained by deleting from P all those statements which do not affect the flow of control of P, i.e., those statements which do not modify control variables or variables which will appear in statements modifying control variables. Moreover in each basic block, i, we add a statement of the form $B_i := B_i + 1$, where the variable B_i does not appear in the original program, has not been used before in P_1 and is associated with the basic block in a unique way. We also add, at the very beginning of P_1, statements which initialize each and every one of these new variables B_i to zero.

(2) The tables $T_1, \dots, T_n$, represent mappings between these names B_i and different kinds of information required for our performance studies. For example, one of these mappings will always be the one which matches the names B_i and the actual sequences of statements which constituted the corresponding basic block.

We shall call P_1 the (flow-of-control) *skeleton* of P.

We notice that in the first clause defining P_1 we need not introduce so many new variables B_i to describe the profile of P, because in any program there is redundancy of flow-of-control information which one can use advantageously [Knu73]. All one needs is to have one variable counting the executions of each independent path. Basic blocks which will always execute sequentially may be accounted for by the use of only one variable. Figure 1.2.1 illustrates a simple instance of this, where we see that the value of B_3 can be derived from (in this case coincides with) the count for B_1.

Discovering which statements affect the flow of control of a program can be done in an automated way by the global data flow analysis methods used for code optimization [Aho79]. The methods essentially consist of a multipass procedure where, in the first pass through the code, data flow information is gathered and suitable data structures that will keep this information for further processing are built. In subsequent passes the information is appropriately used. Several implemented optimizing compilers gather all the information which is needed to perform step (1) of our procedure during their first pass through the code.

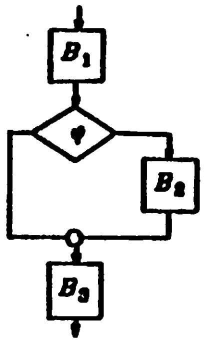

Figure 1.2.1 Two variables B_i suffice.

As for the tables required, some of them can also be generated automatically quite easily. For example, while parsing the program, the compiler can identify all the basic blocks when they are encountered and count the distinct types of atomic operations which appear in each block. However, it should be clear that some other tables require extra effort to obtain. For example, suppose we are interested in determining the page trace of a run of a program; then, we need to know the physical layout of a compiled version of the program to determine which pages correspond to a given basic block. This has to be done after the actual machine code has been produced.

To illustrate the form, effectiveness and usage of the skeleton of a program, as well as its limitations, we shall present three examples. Two of them are taken from a large FORTRAN program and the third is the well known Warshall's algorithm. Warshall's algorithm computes the cost of traversing a labeled directed acyclic graph to go from one vertex to another. When appropriately interpreted, it yields the transitive closure of a matrix.

SPICE is a large FORTRAN program [Coh76b, Nag75] (11000 lines of code) which analyzes integrated circuits to determine their electrical and thermal properties. We have chosen to analyze (parts of) it because it is an example of a large program which is frequently used at Berkeley and whose behavior has been analyzed using only conventional techniques.

For our immediate purposes we have chosen two parts of the code, the subroutine TMPUPD and the subroutine MATLOC, of which we will build the skeletons. The analysis of each one of them will point to advantages and limitations of the method. However, in both cases we will see that obtaining the profile from the skeleton is much faster than from the original code.

Appendix A contains the original FORTRAN code for each of these two portions of SPICE, as well as their flowgraphs in D-chart form.

Example 1.2.1

In Table 1.2.1 we show the code for the skeleton of the subroutine TMPUPD. We see that all the statements are very simple and thus of quick

```
   SUBROUTINE TMPUPD                       IF (ITEMNO.LE.2) GO TO 220
   IMPLICIT DOUBLE PRECISION (A-H,O-Z)     B16=B16+1
   B1=B1+1                              220 B17=B17+1
   IF (ITEMNO.LE.2) GO TO 5                IF (ITEMNO.LE.2) GO TO 230
   B2=B2+1                                 B18=B18+1
 5 B3=B3+1                              230 B19=B19+1
   LOC=LOCATE(1)                           LOC=NODPLC(LOC)
   ITITLE=0                                GO TO 210
10 IF (LOC.EQ.0) GO TO 100              300 LOC=LOCATE(23)
   B4=B4+1                                 B20=B20+1
   LOCV=NODPLC(LOC+1)                      IF (LOC.EQ.0) GO TO 400
   TC1=VALUE(LOCV+3)                       B21=B21+1
   TC2=VALUE(LOCV+4)                    310 IF (LOC.EQ.0) GO TO 400
   IF (TC1.NE.0.0D0) GO TO 20              B22=B22+1
   IF (TC2.EQ.0.0D0) GO TO 40              IF (ITEMNO.LE.2) GO TO 320
20 IF (ITITLE.NE.0) GO TO 30               B23=B23+1
   B5=B5+1                              320 B24=B24+1
   ITITLE=1                                LOC=NODPLC(LOC)
30 B6=B6+1                                 GO TO 310
40 LOC=NODPLC(LOC)                      400 LOC=LOCATE(24)
   B7=B7+1                                 B25=B25+1
   GO TO 10                                IPRNT=1
100 LOC=LOCATE(21)                      410 IF (LOC.EQ.0) GO TO 1000
   B8=B8+1                                 B26=B26+1
   IF (LOC.EQ.0) GO TO 200                 IF(IPRNT.NE.0) B27=B27+1
   B9=B9+1                                 B28=B28+1
110 IF (LOC.EQ.0) GO TO 200                IPRNT=0
   B10=B10+1                               IF (ITEMNO.LE.2) GO TO 415
   IF (ITEMNO.LE.2) GO TO 120              B29=B29+1
   B11=B11+1                            415 B30=B30+1
120 B12=B12+1                              IF (ITEMNO.LE.2) GO TO 420
   LOC=NODPLC(LOC)                         B31=B31+1
   GO TO 110                            420 B32=B32+1
200 LOC=LOCATE(22)                      430 LOC=NODPLC(LOC)
   B13=B13+1                               GO TO 410
   IF (LOC.EQ.0) GO TO 300             1000 RETURN
   B14=B14+1                               END
210 B15=B15+1
```

Table 1.2.1 **Skeleton of TMPUPD**

execution. We could save more processing time by deleting redundant counters and later, when analyzing the results, reconstruct the full profile by taking into account the interdependencies used to eliminate statements from the original code.

In fact, B_3, B_8, B_{13}, B_{20} and B_{25} are always traversed the same number of times as B_1 is. Similarly, B_7 is traversed the same number of times as B_4, B_{12} is traversed the same number of times as B_{10}, B_{17} and B_{19} are traversed the same number of times as B_{15}, B_{24} is traversed the same number of times as B_{22}, and B_{28}, B_{30}, B_{32} are traversed the same number of times as B_{26}.

So we see that from the skeleton depicted we could still eliminate 13 statements of the form Bi=Bi+1 , where i is an integer. We notice that 7 of these statements are within loops, so their deletion certainly increases even more the running time savings.

■

Example 1.2.2

Table 1.2.2 depicts the skeleton for the subroutine MATLOC of SPICE. Its very simple structure permits, if desired, the further elimination of 17 statements of the Bi=Bi+1 type, where i is an integer. Four of these statements are within loops. What we should notice in this example is that, since we have nested loops, the execution time of the skeleton would improve substantially if we could "linearize" them. In fact, from analyzing the code we see that each of the inner nested loops, i.e., those corresponding to variables B_{11}, B_{16}, B_{19} and B_{23}, are traversed NDIM times each time the T-branch of their respective outer loop is taken. NDIM is a variable which is not modified within MATLOC, it is an input value for this subroutine. Linearizing these loops is then a fairly straightforward matter. After the count for the corresponding outer loop is found, one multiplies it by NDIM and obtains the

```
      SUBROUTINE MATLOC
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      B1=B1+1
      LOC=LOCATE(1)
  690 IF (LOC.EQ.0) GO TO 700
      B2=B2+1
      LOC=NODPLC(LOC)
      GO TO 690
  700 LOC=LOCATE(2)
      B3=B3+1
  710 IF (LOC.EQ.0) GO TO 720
      B4=B4+1
      LOC=NODPLC(LOC)
      GO TO 710
  720 LOC=LOCATE(3)
      B5=B5+1
  730 IF (LOC.EQ.0) GO TO 740
      B6=B6+1
      LOC=NODPLC(LOC)
      GO TO 730
  740 LOC=LOCATE(4)
      B7=B7+1
  750 IF (LOC.EQ.0) GO TO 760
      B8=B8+1
      LOC=NODPI/ ,LOC)
      GO TO 750
  760 LOC=LOCATE(5)
      B9=B9+1
  762 IF (LOC.EQ.0) GO TO 764
      B10=B10+1
      DO 763 I=1,NDIM
      B11=B11+1
  763 CONTINUE
      LOC=NODPLC(LOC)
      B12=B12+1
      GO TO 762
  764 LOC=LOCATE(6)
      B13=B13+1
  766 IF (LOC.EQ.0) GO TO 768
      B14=B14+1
      DO 767 I=1,NDIM
      B15=B15+1
  767 CONTINUE
      LOC=NODPLC(LOC)
      B16=B16+1
      GO TO 766
  768 LOC=LOCATE(7)
      B17=B17+1
  770 IF (LOC.EQ.0) GO TO 772
      B18=B18+1
      DO 771 I=1,NDIM
      B19=B19+1
  771 CONTINUE
      LOC=NODPLC(LOC)
      B20=B20+1
      GO TO 770
  772 LOC=LOCATE(8)
      B21=B21+1
  774 IF (LOC.EQ.0) GO TO 780
      B22=B22+1
      DO 775 I=1,NDIM
      B23=B23+1
  775 CONTINUE
      LOC=NODPLC(LOC)
      B24=B24+1
      GO TO 774
  780 LOC=LOCATE(9)
      B25=B25+1
  790 IF (LOC.EQ.0) GO TO 800
      B26=B26+1
      LOC=NODPLC(LOC)
      GO TO 790
  800 LOC=LOCATE(11)
      B27=B27+1
  810 IF (LOC.EQ.0) GO TO 820
      B28=B28+1
      LOC=NODPLC(LOC)
      GO TO 810
  820 LOC=LOCATE(12)
      B29=B29+1
  830 IF (LOC.EQ.0) GO TO 840
      B30=B30+1
      LOC=NODPLC(LOC)
      GO TO 830
  840 LOC=LOCATE(13)
      B31=B31+1
  850 IF (LOC.EQ.0) GO TO 860
      B32=B32+1
      LOC=NODPLC(LOC)
      GO TO 850
  860 LOC=LOCATE(14)
      B33=B33+1
  870 IF (LOC.EQ.0) GO TO 900
      B34=B34+1
      LOC=NODPLC(LOC)
      GO TO 870
  900 LOC=LOCATE(17)
      B35=B35+1
  910 IF (LOC.EQ.0) GO TO 1000
      B36=B36+1
      LOC=NODPLC(LOC)
      GO TO 910
 1000 RETURN
      END
```

Table 1.2.2 **Skeleton of MATLOC**

count for the inner loop.

■

Example 1.2.3

We present in Table 1.2.3 Warshall's algorithm, taken from [Aho74], to illustrate why we must try to find better approaches than building the skeleton representation of programs. We have expressed it in a pseudo Pascal language for better readability.

We see that *all* loops are traversed a fixed number of times and thus, as in Example 1.2.2, linearization of them is not only desirable but possible. It takes a moment's reflection to see that in this case the straight forward skeleton would not run much faster than the actual routine.

■

1.3. Some Related Work

Donald Knuth has pioneered the area of the mathematical analysis of algorithms [Knu71a, Knu71b, Knu78]. In this analysis, for the execution time

```
for i := 1 until n do $C_{ii}^0 := 1 + l(v_i,v_i)$;
for $1 \le i,j \le n$ and $i \ne j$ do $C_{ij}^0 := l(v_i,v_j)$;
for k := 1 until n do
    for $1 \le i,j \le n$ do $C_{ij}^k := C_{ij}^{k-1} + C_{ik}^{k-1} \times (C_{kk}^{k-1}) \times C_{kj}^{k-1}$;
for $1 \le i,j \le n$ do $c(v_i,v_j) := C_{ij}^n$;
```

l is a labeling function between nodes

C_{ij}^k is the sum of the labels of all paths from v_i to v_j such that all vertices on the path, except possibly the end points, are in the set $\{v_1, v_2, \ldots, v_k\}$.

$c(v_i, v_j)$ is the cost from v_i to v_j.

Table 1.2.3 **Warshall's Algorithm**

of a given algorithm or program, one attempts to determine the four quantities

<maximum, minimum, average, standard deviation>.

The fourth quantity refers to the standard deviation of the distribution of execution times around the average. Knuth's frequent contributions to this area have not only been a source of inspiration for many researchers but they have also shown how difficult the analysis may become even for relatively simple algorithms.

In [Knu76] we can see that the complete analysis of a rather simple algorithm may require complex mathematical knowledge and expertise. It then becomes quite clear that analyzing large real-life programs may be an enormous task. The required amount of sophistication and level of reasoning about the program seems to go beyond the current level of what can be automated.

Nevertheless there have been efforts to understand the nature and complexity of this process. In a recent Ph.D. dissertation, Knuth's student Lyle H. Ramshaw [Ram79] axiomatizes a part of the process of analyzing programs. His efforts are directed towards the understanding of the reasoning behind the mathematical analysis of an algorithm.

With a different approach, since 1974 Jacques Cohen and his collaborators have been *microanalysing* structurally simple programs, i.e., determining the above mentioned four quantities as functions of each elementary operation involved in the program. In [Coh74] Cohen presented a system which would accept programs in a restricted Pascal-like programming language and would return an expression of its execution time as a function of the processing time of elementary operations. However, the evaluation of

this expression requires the user to specify the number of times the body of a loop would be traversed and the branching probabilities of conditional statements. These two conditions make this approach very difficult to use when one is trying to *gain* knowledge about the behavior of a program.

Nevertheless, Cohen's interactive system includes many features for the algebraic simplification of expressions, for finding closed forms for some kinds of summations and for solving some finite difference equations. These features have to exist in any system that will perform a task of this kind.

The simple structure of many algorithms has proved that the method can yield interesting results. In [Coh76a] we see an analysis of Strassens's matrix multiplication algorithm. A non recursive version of the algorithm has all loops traversed a fixed number of times and no conditional statements within loops. This allows the authors to find a closed form expression for the processing time of the algorithm whose evaluation does not lead to inconsistencies. In their expression, specifying the number of times a loop is to be traversed is given by the dimension of the matrices. Then, as all the bodies of the loops are basic blocks, the evaluation yields the exact profile of the run.

However, it is not clear from the presentation in [Coh76a] how much of the mathematical deductions were carried out automatically by the system. The article seems to suggest that these deductions were presented to the system for further symbolic processing and evaluation, but had been obtained by the authors.

We shall call Cohen's approach the *deterministic microanalysis* of programs because of the requirement that the user provide the number of times a loop will be executed and a conditional branch will be taken. A big drawback of this method is that, in any relatively complex program, the interrelationships between statements may become very obscure and involved. It is unreasonable to expect that a user will master them and provide consistent data for the evaluation of the expressions. The fact that these expressions do not depend on the input variables of the analyzed algorithm or program appears to be responsible for most of the method's deficiencies.

It is quite easy to see that all programs which are syntactically correct in the language accepted by the analyzing system are deterministically microanalyzable. As the control structures of this language include while loop statements and if then else branching statements, because of the result of Böhm and Jacopini regarding the functional completeness of this class of programs [Böh66], it would be very nice if for such a class of programs one could produce expressions which described the behavior of the program as a function of the input variables and of the elementary operations. We shall see in Chapter 2 that this is impossible to do, even if we assume that our programs halt.

A different approach can be found in Ferrari's work, [Fer78], where programs are viewed as D-charts and formulae are built in a bottom up fashion taking into account all the data dependencies. Unfortunately, the methodology used there did not clarify when one could obtain such expressions. Only very simple examples were found to be manageable. However, the expressions obtained were functions of the input variables and thus when supplied with values for them a correct profile was obtained. The task of finding expressions became more complicated but their evaluation required nothing from the user, and the answer obtained was always correct.

To obtain the four quantities desired using Ferrari's expressions, one has to find suitable input data that would exercise the program in such a way as to achieve its minimum and its maximum; then, making some probabilistic assumptions on the nature of the input data, one is able to determine the average and standard deviation with some predetermined degree of statistical confidence by measuring enough samples of the input data. In fact, it is worth noting that Cohen's approach requires the same kind of hypothesis with the additional problem that, for a given assignment of values to the number of times loops are traversed and branches taken, one may not obtain values which represent the execution of the program under a given set of inputs.

A very interesting system, Metric, is presented in [Weg75]. With it a very limited class of Lisp programs can be correctly microanalyzed. The highlights of Wegbreit's system are that it knows how to find closed form formulae for recursive programs (in its restricted Lisp environment), deals with algebraic simplifications and expresses the execution behavior as a function of the size of the input. Moreover, Metric also allows several measures of performance to coexist. This provides a degree of flexibility that Cohen's system does not have. However, when computing the maximum and minimum execution time of a program, as in Cohen's system, several "simplifying" hypotheses are made which yield bounds not necessarily tight. In other terms, there may be no set of inputs which would make the program attain these bounds.

The very fertile area of Symbolic Evaluation or Symbolic Execution of programs has undisputed relevance to our problem. In [Che79, Che76, Che78, Kin76, How78] we read about different systems which attempt to express in a symbolic way the results of the computations performed by a program. Common to all of them, and to any system which performs such a task, is the problem of dealing with loops. The effect that such a construct has on the value of a variable is central to the analysis in all approaches.

All of these authors are primarily concerned with the correctness of the analyzed programs, although performance is mentioned in a paper by Cheatham [Che79]. Systems like DISSECT [How78] are designed to evaluate FORTRAN programs, EFFIGY [Kin76] is intended to evaluate simple PL/I programs, and the system developed by Cheatham and his collaborators [Che79] to analyze EL1 programs. The output normally consists of an expression describing the effect of a program path on a variable and the sequence of predicates whose truth value uniquely determines the execution path taken. The ability of each of these systems to find such expressions rests in the simplicity of the programs submitted for analysis. The limitations of each methodology are never discussed in a formal way.

However, in [Che79] we find for the first time a decision procedure for solving a restricted class of recurrence relations [Kar79]. The result by Gosper [Gos78] gives us more tools to work with in this area. Gosper's result is implemented in MACSYMA [Mat77], which is a powerful algebraic manipulation system originally implemented at MIT to run on PDP-10's. However, today there is a Berkeley version of MACSYMA ("vaxima"), [Fat79], which runs on VAX computers. MACSYMA as a tool in symbolic evaluation appears to have no rivals. In Chapter 5 we shall deal with the problem of finding closed forms and describe how a system like MACSYMA may help a user doing it.

The only reference known to us that uses an idea similar to the skeleton is in [Pra79], where programs are decomposed into a control part (a subset of our skeleton) and a kernel part (which is only concerned about computing output values). Then the author uses this idea to study program equivalence, termination and code optimization. He introduces several models of programs, the most general of which formalizes the notion of equivalence of control structures. It should be clear that any two programs which share control structures will behave identically in their profile equations, and moreover, from the viewpoint of their termination, one will halt if and only if the whole class of programs with the same control structure halts.

The notion of Data Flowgraphs and Flowcharts [Kod78a, Kod78b] has been used to analyze the behavior of programs. However, there are many problems with this essentially static approach. Some of the problems are well stated in [Dav80], where the author specifically criticizes alleged analogies between flowgraphs and electric circuits, but his solution does not go very far in solving the main objections to the approach. The main problem is that all of the dynamic aspects of the program are totally lost. Devices must be introduced to describe, for example, the number of times a given loop will be traversed. In fact, what is missing, once again, is the dependence of the representation on the input data.

In the following two chapters we shall study families of representations for programs which improve on our skeleton approach (by having a faster running time) and which will always be dependent on the input data. To achieve this we first introduce pertinent formalisms in the next chapter.

CHAPTER 2

A Program Performance Language and its Semantics

We shall now introduce a formal language which will be used to express our symbolic representations of programs. Its spirit is similar to that of languages used in first order logic. However, a symbol which adequately enables us to deal with control structures has been introduced in our language. For a simple introduction to first order logic languages we refer the reader to [End72].

We assume we have an infinite set of symbols which is partitioned as follows:

Logical Symbols

1 parentheses: (,)

2 sentential connective symbols: $\neg$, OR

3 variables (one for each non-negative integer n): $x_0, x_1, \ldots, x_n, \ldots$

4 equality symbol: $=$.

Non Logical Symbols

1 one binary predicate symbol: $<$

2 two constant symbols: 0, 1

3 function symbols: the unary function symbol log, the binary function symbols $+$, $*$, mod, and, for each positive integer n, some sets (possibly empty) of symbols, called n-place function symbols.

4 the four-place special symbol: IFTHENELSEFI .

5 the special denotation symbols (one for each non-negative integer n):

$$B_0, B_1, \ldots, B_n, \ldots$$

The constant symbols are sometimes also called 0-place function symbols. This allows for a uniform treatment when we specify the semantics of the language.

Our intended interpretation of most of these symbols should be quite clear. All the unary and binary operation symbols describe the basic real valued algebraic operations and the constants 0 and 1 are to mean zero and one. The special denotation symbols B_i will be used to represent the basic blocks (of instructions in a program). We shall make all the meanings explicit after we introduce the syntax for the language.

2.1. Program Performance Formulae

An expression is any finite sequence of symbols. The simplest kind of meaningful expressions are the *terms*. They are the expressions which are interpreted as naming numerical objects. The two kinds of objects we are going to be concerned with are the basic blocks (of instructions in a program) and numerical values.

Normally in mathematical logic terms are all those expressions which can be built up from the constant symbols and the variables by prefixing the function symbols. Formally, for each n-place function symbol f, one defines an n-place term-building operation Γ_f on expressions:

$$\Gamma_f(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n) = f(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)$$

and uses it to generate the set of terms. However, we shall adopt a more restricted definition in that not all function symbols will be used to build up our terms.

Definition 2.1.1

The set of *terms* is the set of expressions generated from the constant symbols and variables by the operations $\Gamma_{\log}$, Γ_*, Γ_+ and Γ_{mod}.

■

From now on, whenever we refer to an n-place function symbol f, f will not be one of log, +, * or mod. However, if we say "any n-place function symbol f" then the above four function symbols are also included.

Definition 2.1.2

An *atomic formula* is an expression of the form $P(t_0,t_1)$, where P is either the equality symbol = or the binary relation symbol < and t_0, t_1 are terms. We shall abbreviate atomic formulae by writing $t_0 = t_1$ and $t_0 < t_1$.

■

Informally, sentential formulae are those expressions which can be built up from the atomic formulae by use of the sentential connective symbols. This can be made precise by using the following two sentential-formula-building operators on expressions:

$$\Gamma_{\neg}(\varepsilon) = (\neg\varepsilon),$$
$$\Gamma_{\text{OR}}(\varepsilon_1,\varepsilon_2) = (\varepsilon_1 \text{ OR } \varepsilon_2).$$

Definition 2.1.3

The set of *sentential formulae* (formulae, for short) is the set of all expressions generated from the atomic formulae by the operations $\Gamma_{\neg}$ and Γ_{OR}.

■

Definition 2.1.4

We define our set of program performance formulae by recursion on the length of expressions. We let Λ denote the empty string.

(i) Λ is a program performance formula.

(ii) Special denotation symbols B_0, B_1, ... are program performance formulae.

(iii) If ψ_1 and ψ_2 are two program performance formulae then $\psi_1\psi_2$ is a program performance formula.

(iv) If φ is a formula, f any n-place function symbol, $t_1, \dots, t_n$ terms, and ψ_1, ψ_2 two program performance formulae, then

$$\text{IFTHENELSEFI}(\varphi, f(t_1, \dots, t_n), \psi_1, \psi_2)$$

is a program performance formula.

■

Program performance formulae will be, under suitable conditions to be described later, symbolic expressions for the profile equations of programs as functions of the input variables. Their linear time evaluation cost is what makes them very desirable for performance evaluation studies.

As is customary when describing formal languages the number and kinds of primitive symbols have been kept to a minimum to avoid redundancies. However, to make the language practical, we introduce abbreviations for commonly used relations and operations.

We thus introduce the following three binary relation symbols: ≤, >, ≥. The longest definition in terms of = and < is for >:

$$t_1 > t_2 \text{ iff } \neg((t_1 = t_2) \text{ OR } (t_1 < t_2))$$

where t_1 and t_2 are terms and iff is an abbreviation for "if and only if".

We also introduce the rest of the binary logical connective symbols: &, → and ↔. The exponential function symbol exp(x,y), also denoted as x^y, and the binary division function symbol / (in whose definition we exclude the possibility of dividing by zero) are also defined in terms of our primitive function symbols in the usual way.

From now on the program performance formula

$$\mathrm{IFTHENELSEFI}(\varphi, f(t_1, \dots, t_n), \psi_1, \psi_2)$$

will be written

$$\mathrm{IF}\ \varphi\ ,\ f(t_1, \dots, t_n)\ \mathrm{THEN}\ \psi_1\ \mathrm{ELSE}\ \psi_2\ \mathrm{FI}\ .$$

2.2. Semantics for Program Performance Formulae

We shall now define the (canonical) interpretation of the syntactic objects introduced in the previous section. As no quantifiers exist in our language, all variables which appear in a program performance formula (ppf) are free. This will enable us to evaluate any ppf in a one-pass left-to-right manner. This can not be achieved when quantifiers are present.

As a full (mathematical logic) model theory for this language does not seem to play a role in our problem, we shall not develop it here. Indeed, our (standard) universe will be the set of real numbers, even though there will be cases when some variables will only range over integer values.

Let ι: V → $\mathbb{R}$ be an assignment function from the set V of all variables into the set of real numbers. We define an extension ι of ι to the set of all expressions denoting numerical values as follows:

1 for each variable x, $\iota(x) = \iota(x)$.

2 $\iota(0) = 0$ and $\iota(1) = 1$.

3 If $t_1, \dots, t_n$ are terms and f is one of log, mod, +, *, then

$$\iota(f(t_1, \dots, t_n)) = f^{\mathbb{R}}(\iota(t_1), \dots, \iota(t_n)),$$

where $f^{\mathbb{R}}$ is the operation defined in the real numbers which is denoted by f. In particular, × denotes the multiplication operation, i.e., $*^{\mathbb{R}}$ is × .

4 If $t_1, \dots, t_n$ are terms and f is an n-place function symbol different from log, mod, +, *, then

$$\iota(f(t_1, \dots, t_n)) = f^{\mathbb{R}}(\iota(t_1), \dots, \iota(t_n)),$$

where $f^{\mathbb{R}} : (\mathbb{R} \cup \{\infty\})^n \to \mathbb{R} \cup \{\infty\}$ is such that if any argument is ∞ then the value is ∞.

Having defined the interpretation for terms, we now proceed to define satisfaction for formulae. Given a formula φ and an assignment function ι, $\varphi[\iota]$ is the result of assigning values, via ι, to all (free) variables in φ.

With atomic formulae,

for any two terms t_1 and t_2, $(t_1 = t_2)[\iota]$ is true iff $\iota(t_1)$ is equal to $\iota(t_2)$. $(t_1 < t_2)[\iota]$ is true iff $\iota(t_1)$ is (strictly) less than $\iota(t_2)$.

With sentential formulae φ,

$(\sim \varphi)[\iota]$ is true iff it is not the case that $\varphi[\iota]$ is true. $(\varphi_1 \text{ OR } \varphi_2)[\iota]$ is true iff $\varphi_1[\iota]$ is true or $\varphi_2[\iota]$ is true.

The interpretation of a ppf will yield a (finite) sequence of symbols which is meant to represent the profile of a program when the program is run with the inputs used to evaluate the ppf.

Definition 2.2.1

We define the *interpretation function* I by induction on the complexity of ppf's:

1 for any special denotation symbol B_i, $\mathrm{I}(B_i)[\iota] = 1B_i$.

2 for any performance formula ψ, where ψ is $\psi_1\psi_2$,

$$\mathrm{I}(\psi_1\psi_2)[\iota] = \mathrm{I}(\psi_1)[\iota]\mathrm{I}(\psi_2)[\iota].$$

3 For any formula φ, n-place function symbol f, terms $t_1, \dots, t_n$ and ppf's ψ_1, ψ_2,

$$I(\text{IF } \varphi \,.\, f(t_1, \ldots, t_n) \text{THEN } \psi_1 \text{ ELSE } \psi_2 \text{ FI})[\iota] \qquad \text{is equal to}$$

$$f^{\mathbb{R}}(\iota(t_1), \ldots, \iota(t_n)) \times I(\psi_1)[\iota] \qquad \text{if } \varphi[\iota] \text{ is true} \quad \text{and equal to}$$

$$f^{\mathbb{R}}(\iota(t_1), \ldots, \iota(t_n)) \times I(\psi_2)[\iota] \qquad \text{if } \varphi[\iota] \text{ is false,}$$

where for any $x \in \mathbb{R} \cup \{\infty\}$, $\infty \times x = x \times \infty = \infty$. If ψ_j is Λ, for $j \in \{1,2\}$, then we say that $f^{\mathbb{R}}(\iota(t_1), \ldots, \iota(t_n)) \times I(\psi_j)[\iota]$ is Λ.

■

Proposition 2.2.1

Let ψ be a program performance formula and ι an assignment function of V into $\mathbb{R}$. Then $I(\psi)[\iota] = s_0 s_1 \ldots s_n$, where (a) $s_0 \in \mathbb{R} \cup \{\infty\}$, (b) for $0 < i \leq n$, $s_i \in \mathbb{R} \cup \{\infty\} \cup \{B_i\}_{i \in \omega}$, (c) if $s_i \in \mathbb{R} \cup \{\infty\}$ then $s_{i+1} \in \{B_i\}_{i \in \omega}$.

Proof

By induction on the complexity of program performance formulae.

■

2.3. Representation of Programs

We shall first study non recursive *goto*-less programs. It is well known [Böh66] that any computation can be carried out by a program of this kind. As done in [Fer78], we shall represent this kind of programs by single-entry single-exit directed graphs called D-charts.

A D-chart has five types of vertices and three rules of formation. The five types of vertices are: *rectangular boxes*, which are used to represent basic blocks of statements in series or more complicated D-charts; *diamond shaped* vertices, which represent decisions; *circular* vertices, which represent junctions; and the two *triangular* vertices, representing entry and exit points. The rules of formation are: composition, alternation, and iteration.

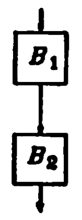

Figure 2.3.1

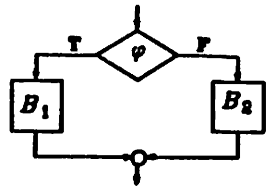

Figure 2.3.2

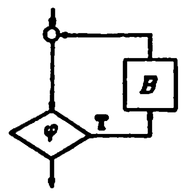

Figure 2.3.3

Definition 2.3.1

(i) If $\boxed{B}$ represents a basic block of statements in a program, then $\boxed{B}$ is an elementary D-chart.

(ii) (Composition) If $\boxed{B_1}$ and $\boxed{B_2}$ are elementary D-charts, then Figure 2.3.1 is an elementary D-chart.

(iii) (Alternation) If $\boxed{B_1}$ and $\boxed{B_2}$ are elementary D-charts and φ a formula (see Def. 2.1.2) then Figure 2.3.2 is an elementary D-chart. We call the two branches the T-branch and the F-branch respectively. For example in Figure 2.3.2 we have the left branch as the T-branch and the right branch as the F-branch.

(iv) (Iteration) If $\boxed{B}$ is an elementary chart and φ a formula, then Figure 2.3.3 is an elementary D-chart. We call the two branches the T-branch and the F-branch respectively. In Figure 2.3.3 the T-branch is the right branch and the F-branch is the down branch. The T-branch of an iteration will always be the "loop back" branch.

Definition 2.3.2

A D-chart is a graph of the form $\boxed{B}$, where $\boxed{B}$ is an elementary D-chart.

Given an elementary D-chart $\boxed{B}$ we shall distinguish two points in it: the *entry point* α and the *exit point* β, which are located just before entering the rectangle B and just after exiting the rectangle B (see Figure 2.3.4). A *check point* γ in an elementary D-chart D is any entry or exit point of an elementary D-chart D' contained in D.

Each path through a D-chart corresponds to a possible flow of control, or *run*, through the original program. In alternations and iterations, the T-branch is taken if the evaluation of the formula φ is true. Otherwise the F-branch is taken. Runs begin with the first statement of a program, with the triangular vertex representing the entry point (assumed to be unique).

The *input variables* of a run are variables which are referenced in the path before they are assigned values. The *explicit control variables* of a run

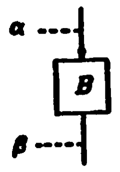

Figure 2.3.4

are those variables which occur in at least one predicate (formula) of an alternation or iteration in the path. A run *halts* if, given the set of input variables, the corresponding execution terminates. We say that a program halts if all of its runs halt.

2.4. Program Performance Formulae for D-charts

We shall now associate in a unique way ppf's to D-charts by inductively assigning ppf's to the basic components of elementary D-charts.

Given a D-chart D, the ppf ψ_D associated with D is obtained as follows:

(1) For each indecomposable elementary D-chart $\boxed{B}$, (i.e., $\boxed{B}$ represents a basic block of instructions), we assign to the basic block a special denotation symbol B_i (never to be used again for any other basic block) and the ppf $1B_i$ to the elementary D-chart.

(2) If $\boxed{B_1}$ and $\boxed{B_2}$ are elementary D-charts with assigned ppf's ψ_1 and ψ_2 respectively, then $\psi_1\psi_2$ is the ppf assigned to their composition.

(3) Given an alternation construct where D_1 and D_2 are the elementary D-charts associated with the T and F branches respectively and φ is the predicate, the ppf associated with it is

$$\text{IF } \varphi \text{ . } 1 \text{ THEN } \psi_1 \text{ ELSE } \psi_2 \text{ FI.}$$

where ψ_1 and ψ_2 are ppf's associated with D_1 and D_2 respectively, and 1 represents the real valued constant function whose value is 1, i.e.: $1(x) = 1$ for all $x \in \mathbb{R}$.

(4) Given an iteration construct D where D_1 is the elementary D-chart associated with the T-branch and φ is the predicate having n variables, the ppf associated with it is

$$\text{IF } \varphi \text{ . } f \text{ THEN } \psi_1 \text{ ELSE } \Lambda \text{ FI .}$$

where ψ_1 is the ppf associated with D_1 and f is an n-place function symbol with the same variables as φ which, when evaluated with the value of the variables at the entrance point a, yields the number of consecutive times that φ would evaluate to true in the corresponding run. We shall denote such a function f associated with φ by $\#\varphi$.

If D' is the elementary D-chart obtained from D by removing the two triangular vertices and ψ' is the ppf associated with D' by the above rules, then ψ' is ψ_D.

Theorem 2.4.1

Assume that P is a program represented by the D-chart D, and ψ_D has $\bar{x}$ as variables. Then P halts iff, for all assignment functions, ι, ∞ does not appear in $I(\psi_D(\bar{x}))[\iota]$.

Proof

P will not halt iff a run enters an iteration and never exits it. We also have that for any program and for each iteration with predicate φ, $\#\varphi$ will have ∞ in its range iff there is a set of inputs which makes the run enter the iteration and never exit it. We finally notice that in the evaluation of a ppf the only place where ∞ can be introduced is when evaluating $\#\varphi$ for some predicate φ.

Thus, if P halts, for no assignment ι will any $\#\varphi$ evaluate to ∞ and so ∞ will not appear in $I(\psi)[\iota]$. Conversely, if ∞ never appears for any assignment ι, then no $\#\varphi$'s ever evaluate to ∞ and so all runs terminate.

■

From now on, we shall assume that our programs halt.

Given a D-chart D and an assignment function ι for its input variables $\bar{x}$, the sequence

$$a_0B_0a_1B_1 \cdots a_nB_n$$

is the *profile* of P under input ι iff, for $0 \le j \le n$, the run with inputs ι traverses a_j times the elementary block of instructions represented by B_j.

Given a D-chart D, ψ_D *represents* the profile equations of P if, for every assignment ι, $I(\psi_D)[\iota]$ is the profile of P under input ι. If ψ_D represents the profile equations of P, we denote it by ψ_P.

We shall now determine conditions on D-charts under which $\psi_D = \psi_P$. There is one construct which presents no problem: composition. If ψ_D is $\psi_{D_1}\psi_{D_2}$ and $\psi_{D_1} = \psi_{P_1}$, $\psi_{D_2} = \psi_{P_2}$, then as $I(\psi_D) = I(\psi_{D_1})\, I(\psi_{D_2})$ we immediately have $\psi_D = \psi_P$. Moreover, alternations and iterations where all the elementary D-charts appearing are basic blocks, also represent the profile equations of their corresponding programs. In the case of iterations this is guaranteed by the definition of $\#\varphi$.

Problems arise with the nesting of non primitive constructs.

Theorem 2.4.2

For any elementary D-chart D where there are neither alternations nor iterations within an iteration, $\psi_D = \psi_P$.

Proof

We prove it by induction on the kind of permissible constructs. Let ι be an assignment function.

Clearly for an elementary D-chart of the form $\downarrow\boxed{B}\downarrow$ where $\boxed{B}$ is an indecomposable elementary D-chart, $\psi_D = \psi_P$ ($= 1B_\iota$ where B_ι is the special denotation symbol assigned to B). Assume now that D_1 and D_2 are two elementary D-charts satisfying our hypothesis for which $\psi_{D_1} = \psi_{P_1}$ and $\psi_{D_2} = \psi_{P_2}$.

Composing them we already know preserves representability. Say we have an alternation with φ as predicate, D_1 as T-branch and D_2 as F-branch. The ppf ψ which represents it is

$$\text{IF } \varphi \text{ . 1 THEN } \psi_{D_1} \text{ ELSE } \psi_{D_2} \text{ FI.}$$

and $I(\psi)[\iota]$ is $1 \times I(\psi_1)[\iota]$ if $\varphi[\iota]$ is true, and is $1 \times I(\psi_2)[\iota]$ if $\varphi[\iota]$ is false. Thus, in either case, we obtain that $\psi_D \equiv \psi_P$ because of our induction hypothesis on D_1 and D_2. As for iterations, our hypothesis only allow them to have basic blocks as T-branches and for these we know they represent the profile equations.

■

Lemma 2.4.3

Let D be an elementary D-chart and, in particular, an iteration with predicate φ_0. Let D's body consist of an alternation D_1 with predicate φ_1, T-branch D_{11} and F-branch D_{12}, where $\psi_{D_{11}} \equiv \psi_{P_{11}}$ and $\psi_{D_{12}} \equiv \psi_{P_{12}}$. (see Figure 2.4.1). Then, $\psi_{D_1} \equiv \psi_{P_1}$ iff the same branch of the alternation is traversed each time the T-branch of the iteration is traversed.

Proof

Let ι be an assignment function. We prove the "only if" part first.

If $\varphi_0[\iota]$ is true, the T-branch of the iteration is traversed $\#\varphi_0^R[\iota]$ times. So if, say, D_{11} is always traversed, the correct profile is given by $\#\varphi_0^R[\iota] \times I(\psi_{D_{11}})[\iota]$. As by hypothesis $\psi_{D_{11}} \equiv \psi_{P_{11}}$, we have $\psi_{D_1} \equiv \psi_{P_1}$ in this case. Similarly, if D_{12} were the branch always traversed, using $\psi_{D_{12}} \equiv \psi_{P_{12}}$ we would obtain $\psi_{D_1} \equiv \psi_{P_1}$.

Now for the "if" part, we argue as follows. The ppf ψ corresponding to D is

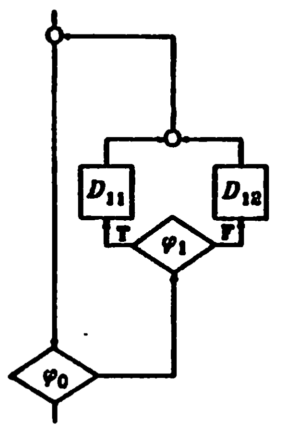

Figure 2.4.1

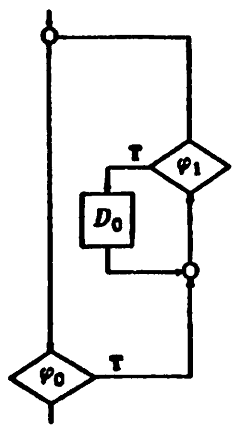

Figure 2.4.2

$$\text{IF } \varphi_0 \text{ . } \#\varphi_0 \text{ THEN } \psi_{D_1} \text{ ELSE } \Lambda \text{ FI.}$$

Thus, $I(\psi)[\iota]$ is $\#\varphi_0^R[\iota] \times I(\psi_{D_{11}})[\iota]$ if $\varphi_0[\iota]$ is true, and Λ if $\varphi_0[\iota]$ is false. We then see that $\psi_D \equiv \psi_P$ implies that the same branch of the alternation is taken each time the T-branch of the iteration is taken.

■

Lemma 2.4.4

Let D be an elementary D-chart and, in particular an iteration with predicate φ_0. Let D's body consist of another iteration with predicate φ_1 and body D_0. (see Figure 2.4.2). We assume further that $\psi_{D_0} \equiv \psi_{P_0}$. Then, $\psi_D \equiv \psi_P$ iff there exists an integer n such that each time the T-branch of the outer iteration is traversed the T-branch of the inner iteration is traversed n times.

Proof

First we deal with the "only if" part.

If $\varphi_0[\iota]$ is true, the T-branch of the outer iteration is traversed $\#\varphi_0^R[\iota]$ times. If $\varphi_1[\iota]$ is true and we are traversing the outer loop's T-branch, then the inner one will be traversed $\#\varphi_1^R[\iota]$ times. Thus, $\#\varphi_0^R[\iota] \times \#\varphi_1^R[\iota] \times I(\psi_{D_{11}})[\iota]$ is the correct profile for D in this case, since $\psi_{D_{11}} = \psi_{P_{11}}$ and since the inner iteration will always traverse the same number of times its T-branch each time the outer iteration's T-branch is traversed. If φ_1 is false, the evaluation yields $\#\varphi_0^R[\iota] \times \Lambda \times I(\psi_{D_{11}})[\iota]$, which is equal to Λ by our definition. So in either case we see that $\psi_D = \psi_P$. The last case is when φ_0 is false but then Λ is again the correct answer.

Now we deal with the "if" part. The ppf ψ corresponding to D is

$$\text{IF } \varphi_0 \,.\, \#\varphi_0 \text{ THEN IF } \varphi_1 \,.\, \#\varphi_1 \text{ THEN } \psi_{D_0} \text{ ELSE } \Lambda \text{ FI ELSE } \Lambda \text{ FI.}$$

The two interesting cases of $I(\psi)[\iota]$ are when both $\varphi_0[\iota]$ and $\varphi_1[\iota]$ are true, and when $\varphi_0[\iota]$ is true and $\varphi_1[\iota]$ is false. In the latter case $I(\psi)[\iota]$ is Λ, and so this forces the inner iteration to satisfy the condition that, if φ_1 was false the first time the T-branch of the outer iteration was traversed, then φ_1 will remain false for all consecutive traversals.

If $\varphi_0[\iota]$ and $\varphi_1[\iota]$ are both true, then $I(\psi)[\iota]$ is $\#\varphi_0^R[\iota] \times \#\varphi_1^R[\iota] \times I(\psi_{D_0})[\iota]$, which represents the profile equations of D only if $\psi_{D_0} = \psi_{P_0}$ and the inner iteration's T-branch is always traversed the same number of times each time the T-branch of the outer iteration is taken. This number of times corresponds to that traversed the first time the outer iteration's T-branch was taken.

■

It is worth mentioning that even though the last two Lemmas are quite discouraging, in our Examples 1.3.1, 1.3.2 and 1.3.3 all nested loops satisfy the hypothesis of Lemma 2.4.4. In fact, as noted in Section 1.3, the hypotheses of the Lemmas 2.4.3 and 2.4.4 have been implicitly made in all the literature we know about.

As for the hypotheses of Lemma 2.4.3, they are only relevant in our first example and there they hold for six of the eight existing cases. In the two cases in which they fail, one branch is taken the first time the T-branch of the iteration is taken (to print headlines) and the other branch is taken for all successive traversals.

In a D-chart, whenever alternations within iterations satisfy the hypothesis of Lemma 2.4.3, we say that *alternations are well behaved*. Similarly if nested iterations satisfy the hypothesis of Lemma 2.4.4 we say that *iterations are well behaved*.

Theorem 2.4.5 Representability of Profile Equations

$\psi_D = \psi_P$ iff for all assignments ι all alternations and iterations are well behaved.

Proof

Let us deal first with the "only if" part. The proof is by induction on the complexity of elementary D-charts. Clearly the symbols B_i represent the profile of the basic blocks of instructions which they are associated to. We have already remarked that

alternations and iterations with irreducible D-charts as branches represent the profile of their associated programs. The other two building steps make use of Lemma 2.4.3 or Lemma 2.4.4, and the hypothesis that alternations and iterations are well behaved. Thus, whenever a possible conflicting construct occurs, i.e., an alternation or an iteration within an iteration, our well behavedness hypothesis allows us to conclude that we still represent the profile equations of the larger elementary D-chart.

Now we deal with the "if" part. As in the proofs of Lemmas 2.3.3 and 2.3.4, we must analyze the effect $\psi_D = \psi_P$ has on D-chart constructs. We only have to look at two cases: alternations within iterations and iterations within iterations, because the all other cases cause no problems.

Assume we have an alternation D_1 with predicate φ_1 within an iteration D with predicate φ_0. The iteration may be located as in Figure 2.4.1 or there may exist an elementary D-chart between the entrance of the alternation and the entrance of the T-branch of the iteration. By the composition rule, in any of these two cases we will have that when evaluating the ppf corresponding to D, $\#\varphi_0^{R}[t] \times I(\psi_{D_1})[t]$ will appear if $\varphi_0[t]$ is true. But then, this is as in Lemma 2.4.3, so we must have that this alternation is well behaved. In the same way we argue that all alternations in the D-chart must be well behaved.

Similarly, using the proof of Lemma 2.4.2, we argue that all iterations appearing in the D-chart must be well behaved.

■

Theorem 2.4.5 shows that our goal of Section 1.1, that of obtaining ppf's representing profile equations evaluable efficiently, i.e., in a one-pass left-to-right procedure in linear time (as a function of the number of characters in the ppf), forces rather strong topological and/or semantic constraints on the programs that these ppf's represent.

A natural question to ask then is whether this is due to our inability to formalize the problem or to an essential characteristic of computations which does not allow us to "linearize" all of them. What would seem to be missing in our evaluation procedure is a way of taking into account the interdependencies between control variables of nested constructs. Perhaps we might gain from trying to capture more semantics in I. Our assertion is that there is not much more that one can do in full generality, and thus complicating I is not worth it. In Chapter 3 we present an approach which we think is the most appropriate to deal with this problem.

Example 2.4.1

In Figure 2.4.3 we show the D-chart corresponding to a program which reads an array A of N numbers and then stores in S the sum of all the positive entries of the array. Thus, in Figure 2.4.3 φ_0 is $i \leq N$ and φ_1 is $A[i] \geq 0$. In this example we see that the selection of the branch in the alternation is exclusively dependent on the input data, and that, in order to establish the correct profile for a run, one has to read all input values and compute the cardinality of the sets of "trues" and "falses" of the inner predicate. Thus, even though we know that the T-branch of the iteration will be traversed exactly N consecutive times, one has to evaluate the alternation predicate each time. This precludes the evaluation in linear time of the ppf of this D-chart.

■

Example 2.4.2

Figure 2.4.4 depicts the flowchart of a program which reads an N by M array of numbers A and then adds all the positive elements in the jth column up to the A[1,j]th one in the jth entry of the array S. Thus, the outer predi-

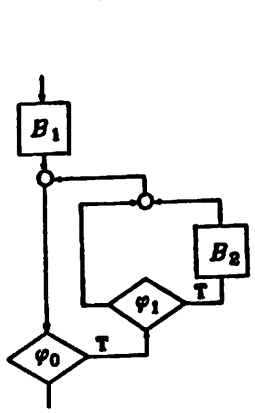

Figure 2.4.3

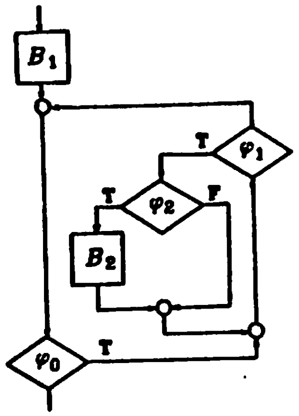

Figure 2.4.4

cate, φ_0, is $i \le M$, φ_1 is $j \le A[1,i]$ and φ_2 is $A[i,j] \ge 0$. In this example we see that both the number of times the inner loop will be traversed and which branch of the alternation is to be taken are absolutely input data dependent. φ_1 and φ_2 have to be evaluated every time. Thus, there can be no purely syntactic interpretation function which can capture this behavior.

■

Let $L = \langle B, \varphi, \alpha, \beta\rangle$ be an iteration with T-branch body B, predicate φ, entry point α and exit point β. Let x be a variable appearing in the D-chart. We denote by $x[\alpha]$ the value of the variable x at the entry point of the iteration; by $x[\beta]$ its value at the exit point of the iteration and by $x[k]$ the value it has immediately after the kth traversal of the T-branch B; note that $x[0]$ is assumed to be $x[\alpha]$, i.e., the value of x at the entrance to the iteration. We extend this notation in the natural way to n-tuples of variables $\mathbf{x}$; thus $\mathbf{x}[\alpha]$

abbreviates $\langle x_1[\alpha], \ldots, x_n[\alpha]\rangle$. k, when used as above, will be called the *iteration index*.

The hypothesis of Lemma 2.4.3 can also be characterized by a logical condition on the predicates φ_0 and φ_1:

Theorem 2.4.6

Let D be an iteration with predicate φ_0, whose body consists of an alternation D_1 with predicate φ_1, T-branch $\psi_{D_{11}}$ and F-branch $\psi_{D_{12}}$. (see Figure 2.4.1). Then, the same branch of the alternation is traversed each time the T-branch of the iteration is traversed iff for all assignment functions ι, whenever the T-branch of the iteration is traversed, (1) $(\varphi_0(\mathbf{x}_0[\alpha]) \rightarrow \varphi_1(\mathbf{x}_1[\alpha]))$ true implies $(\varphi_0(\mathbf{x}_0[k]) \rightarrow \neg\varphi_1(\mathbf{x}_1[k]))$ is false for all positive integers $k \le \#\varphi_0^R(\mathbf{x}_0[\alpha])$ or (2) $(\varphi_0(\mathbf{x}_0[\alpha]) \rightarrow \neg\varphi_1(\mathbf{x}_1[\alpha]))$ true implies $(\varphi_0(\mathbf{x}_0[k]) \rightarrow \varphi_1(\mathbf{x}_1[k]))$ is false for all positive integers $k \le \#\varphi_0^R(\mathbf{x}_0[\alpha])$.

Proof

Let ι be an assignment function. We prove the "if" part first.

Given that the same branch of the alternation is traversed each time the T-branch of the iteration is traversed, then exactly one of (1) or (2) is true. Indeed, if the T-branch of the alternation is traversed, then $(\varphi_0(\mathbf{x}_0[k]) \rightarrow \varphi_1(\mathbf{x}_1[k]))$ would be true for all integers k, $0 \le k \le \#\varphi_0^R(\mathbf{x}_0[\alpha])$. Thus, $(\varphi_0(\mathbf{x}_0[k]) \rightarrow \neg\varphi_1(\mathbf{x}_1[k]))$ would be false for all integers k, $0 \le k \le \#\varphi_0^R(\mathbf{x}_0[\alpha])$.

For the "only if" part we argue as follows. Say (1) is true. Then $(\varphi_0(\mathbf{x}_0[k]) \rightarrow \neg\varphi_1(\mathbf{x}_1[k]))$ being false for all positive integers $1 \le k \le \#\varphi_0^R(\mathbf{x}_0[\alpha])$ means that the F-branch of the alternation is never taken if the T-branch has been taken the first time. We argue in an analogous manner if (2) is true. So, for the run which corresponds to the inputs ι, a unique branch of the alternation will always be traversed each time the T-branch of the iteration is traversed.

■

The advantage of this new characterization is its syntactic orientation. One can now hope that with the aid of a theorem prover, this condition could be checked during a syntactic analysis of the code. In fact, if the predicates φ_0 and φ_1 are of the form $x \mathrm{RO} y$, where RO is one of $<$, $\leq$, $>$ or $\geq$, some cases (depending on the action of the iteration on the control variables) can be analyzed automatically without much difficulty.

2.5. Definable Programs

We now deal with the necessary and sufficient conditions to obtain ppf's which represent the profile equations and in which no n-place function symbols f appear. These ppf's will be symbolic expressions for the profile equations of programs.

The set of syntactical objects which denote numerical values needs to be expanded so as to reflect the effect of alternations and iterations on variables. This amounts to formalizing the symbolic evaluation of program variables.

Definition 2.5.1

The set of *special terms* is defined by recursion on the length of expressions by the following clauses:

1 any term t is a special term;

2 if $\tau_1 \dots \tau_{n+m+2}$ are special terms, $\varphi(x_1, \dots, x_n)$ a formula (Def. 2.1.2) and f an m-place function symbol, then

$$\text{IF } \varphi(\tau_1, \dots, \tau_n) \cdot f(\tau_{n+1}, \dots, \tau_{n+m}) \text{ THEN } \tau_{n+m+1} \text{ ELSE } \tau_{n+m+2} \text{ FI}$$

is a special term;

3 the set of special terms is closed under the operations Γ_{log}, Γ_{mod}, $\Gamma_{\bullet}$ and Γ_{+}.

■

Special terms can be evaluated using the same interpretation function I introduced in Section 2.2.

Proposition 2.5.1

For any special term τ and assignment function ι, $I(\tau)[\iota] \in \mathbb{R} \cup \{\infty\}$.

Proof

By induction on the complexity of special terms.

■

We notice, as in Theorem 2.4.1, that a special term τ evaluates to ∞ iff some $\#\varphi$ appearing in τ evaluates to ∞. Thus, when dealing with halting programs, the evaluation of any special term is finite (recall that our definition of $/$ does not allow division by zero).

Definition 2.5.2

Given an iteration $L = \langle B, \varphi(\bar{x}), \alpha, \beta \rangle$, we say that L is *definable* if there exists a special term $\hat{\varphi}(\bar{x})$ which does not contain n-place function symbols f, such that, if $\varphi(\bar{x}[\alpha])$ is true, then

$$\hat{\varphi}(\bar{x}[\alpha]) = \#\varphi^{\mathbf{R}}(\bar{x}[\alpha]) .$$

■

This last definition is central in what follows. $\hat{\varphi}$ is nothing else but an effective description of $\#\varphi$. When evaluated it yields, as a function of the values of the control variables at the entrance of the iteration L, the number of consecutive times that the T-branch of L will be traversed.

We shall deal later with the important problem of automatic recognition and construction of special terms $\hat{\varphi}$ directly from the syntax of programs. Now, we remark that there may not be a simple relationship between the values of $\hat{p}$ and $\neg\hat{p}$. Their ranges of validity are disjoint.

Control variables will henceforth be assumed to be of numeric type, i.e., type integer or type real. We also assume that the basic operations which can be performed (by our programming languages) on variables are: subtraction, addition, multiplication, division, exponentiation, modulo arithmetic, logarithm evaluation and nth root extraction. It should be clear that our set of terms suffices to represent each one of these actions on variables. For example, if the assignment statement $x_i := x_j * x_i$ occurs in a basic block, then the term $x_j * x_i$ represents it.

We now want to define expressions representing variables which, when evaluated with the values of the input variables at a check point γ, will yield the value of the variable which they represent at γ. Moreover we want these expressions to be special terms. This last requirement forces a constraint common to all systems dealing with symbolic evaluation: that there be a way of expressing (in whatever formal language is used) the effect of an iteration on a variable.

Example 2.5.1

Assume that we have an iteration such that in its T-branch D_1 the variable x is only modified by the assignment $x := a*x + b$, where a and b are names of variables whose value does not change in D_1, and $a[0] \neq 1$. Then $x[k]$ can be expressed by $a^k x[0] + b\left(\frac{a^k - 1}{a - 1}\right)$.

■

The algebraic expression for $x[k]$ depicted in the above example plays a central role in Chapter 4.

Definition 2.5.3

For any variable x, special term τ and check point γ, we say that τ *describes* x at γ if, for any assignment function ι , $I(\tau)[\iota]$ is equal to $x[\gamma]$ in the corresponding run.

■

Definition 2.5.4

For any variable x and any iteration L we say that $\tau(x,y)$ is a *closed form* for x in L if $\tau(x,y)$ is a special term such that for any integer k , $\tau[x[0],k]$ is equal to $x[k]$.

■

We notice that, without loss of generality, y can be assumed to be of type integer. Example 2.5.1 depicts a closed form for x in the associated iteration L . Determining the existence of closed forms will be the subject of Chapter 5.

For the rest of this chapter, we shall assume that closed forms exist for every variable and iteration we consider.

Definition 2.5.5

For any D-chart D, variable x_j, and check point γ, the *canonical special term* (cst) $\tau_{j,\gamma}$ associated with x_j at γ is defined as follows:

(1) If x_j is an input variable at γ of an elementary D-chart with entrance point α, then $\tau_{j,\gamma} = \tau_{j,\alpha}$. Moreover, if the entrance point is the entrance to the D-chart D, then $\tau_{j,\gamma} = x_j$;

(2) for any irreducible elementary D-chart, see Figure 2.5.1, $\tau_{j,\beta}$ is symbolically expressed in terms of $\tau_{j,\alpha}$ as the term representing the result of the sequence of symbolic evaluations of the assignment statements to x_j occurring in B, where x_j is used to denote $\tau_{j,\alpha}$;

(3) for any alternation, see Figure 2.5.2, $\tau_{j,\beta}$ is

IF φ . 1 THEN $\tau_{j,\beta_1}\{\alpha_1/\alpha\}$ ELSE $\tau_{j,\beta_2}\{\alpha_2/\alpha\}$ FI

where, for $k \in \{1,2\}$, $\tau_{j,\beta_k}\{\alpha_k/\alpha\}$ is the expression obtained by replacing in τ_{j,β_k} each occurrence of x_j by $\tau_{j,\alpha}$;

(4) for every iteration, see Figure 2.5.3, $\tau_{j,\beta}$ is

IF φ . 1 THEN $\tau(\tau_{j,\alpha}, \#\varphi)$ ELSE $\tau_{j,\alpha}$ FI

where $\tau(x_j, y_j)$ is a closed form for x_j in L.

■

So, for any elementary D-chart D, any variable x_j and any checkpoint γ in D, the cst $\tau_{j,\gamma}$ exists iff there exist closed forms for x_j in all intervening iterations.

Theorem 2.5.2

For any elementary D-chart D, any variable x_j and any checkpoint γ in D, if the cst $\tau_{j,\gamma}$ exists, then it describes x_j at γ.

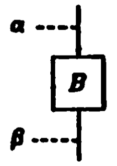

Figure 2.5.1

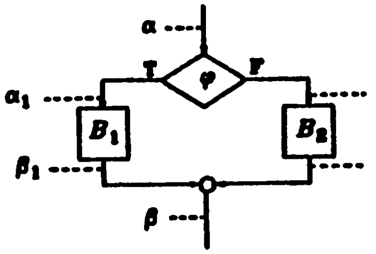

Figure 2.5.2

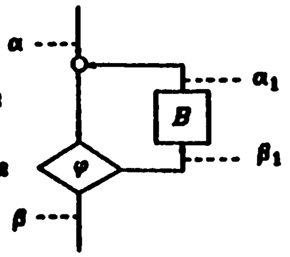

Figure 2.5.3

Proof

By induction on the complexity of cst's, where for the iteration construct (the hard one) we use our assumption that closed forms exist and that they describe their associated variable.

■

As compositions and alternations preserve cst's, we see that all the complexity in building them from simpler ones lies in iterations. It is the nonexistence of closed forms which limits our ability to generate cst's.

In general, making the assumption that a closed form exists for an irreducible iteration is the same as requiring that a given recurrence equation has symbolic solution [Che79] (see Section 5.1). When we assume no nested iterations, as we are allowing alternations within iterations, one has two basically different cases: (i) When considering D-charts for which $\psi_D = \psi_P$, then the existence of the closed form reduces to finding solutions for each of the possible paths and then determining, based on the sole analysis of the input variables, which path will be taken and thus which solution to use for the run. (ii) In the general case where distinct branches may be taken within the same run, no method is known for finding closed forms. In fact, the closed form in case (i) will in general have to allow conditional statements to reflect the fact that distinct branches may be traversed. These issues are discussed in Section 5.4.

If, for a check point γ in a D-chart D, no n-place function symbol f appears in $\tau_{j,\gamma}$, we say that x_j is *definable* at γ and abbreviate this by saying that $\tau_{j,\gamma}$ is definable. We say that an elementary D-chart D preserves definability for x_j if, whenever $\tau_{j,\alpha}$ is definable, then $\tau_{j,\beta}$ is also definable.

Theorem 2.5.3 Definability Preservation for Variables

(i) If $\tau_{j,\alpha}$ is definable and D is an irreducible elementary D-chart, then $\tau_{j,\beta}$ is definable.

(ii) If $\tau_{j,\alpha}$ is definable, D is an alternation in which both the T-branch and the F-branch preserve x_j's definability, and φ is such that each of its control variables is definable at α, then $\tau_{j,\beta}$ is definable.

(iii) If $\tau_{j,\alpha}$ is definable, D is a definable iteration (Def. 2.5.2) with predicate φ such that each of its control variables is definable at α, and there exists a closed form for x_j, then $\tau_{j,\beta}$ is definable.

Proof

(i) This is clear, because it just amounts to having terms representing the basic operations performed on variables, and using $\tau_{j,\alpha}$ as the description of x_j at α.

(ii) As in Figure 2.5.2, let α_1 and α_2 denote the entrance to the T-branch and F-branch of the alternation respectively, and β_1 and β_2 denote the corresponding exits. By assumption, τ_{j,β_1} and τ_{j,β_2} are definable if τ_{j,α_1} and τ_{j,α_2} are. But, in any alternation, $\tau_{j,\alpha} = \tau_{j,\alpha_1} = \tau_{j,\alpha_2}$, so τ_{j,β_1} and τ_{j,β_2} are definable. Then, $\tau_{j,\beta}$ is

$$\text{IF } \varphi \text{ . 1 THEN } \tau_{j,\beta_1} \text{ ELSE } \tau_{j,\beta_2} \text{ FI}$$

and, as each of the control variables is definable at α, $\tau_{j,\beta}$ is definable.

(iii) For any assignment ι, $I(\tau_{j,\beta})[\iota]$ is equal to $x_j[\beta]$ in the corresponding run, but this is equal to $x_j[\#\varphi^{R}[\iota]]$, where $x[0]$ is $I(\tau_{j,\alpha})[\iota]$. Since we have a closed form for x_j in the iteration, $\tau_{j,\beta}$ is $\tau(\tau_{j,\alpha},\#\varphi)$, and, since our iteration is definable, we can express this by $\tau(\tau_{j,\alpha},\hat{\varphi})$. As each of the control variables is definable at α, we can obtain our expression for $\tau_{j,\beta}$ with no n-place function symbols f appearing and thus $\tau_{j,\beta}$ is definable.

■

Each of the converses to (i), (ii) and (iii) in Theorem 2.5.3 deserves individual attention because none of them holds in full generality. For example, definability may be quite easily *regained* after an irreducible elementary D-chart D: just consider the case where the variable is assigned a constant value in D. Thus, the relationship of definability between $\tau_{j,\beta}$ and $\tau_{j,\alpha}$ is not as direct as the one between $\tau_{j,\alpha}$ and $\tau_{j,\beta}$.

Theorem 2.5.4 Definability Acquisition for Variables

(i) If D is an irreducible elementary D-chart in which assignments to x_j are independent of $x_j[\alpha]$ and are either based on input variables, constant values or definable variables, then $\tau_{j,\beta}$ is definable.

(ii) If D is an alternation where (1) $x_j[\beta_1] = x_j[\beta_2]$ and $x_j[\beta_1]$ and $x_j[\beta_2]$ are independent of $x_j[\alpha]$ and definable, or where (2) $x_j[\beta_1] \neq x_j[\beta_2]$, independent of $x_j[\alpha]$, definable and the alternation predicate φ is such that all of its control variables are definable at α, then $\tau_{j,\beta}$ is definable.

(iii) If D is an iteration, k an iteration index, and (1) $x_j[k]$ is independent of $x_j[\alpha]$ and constant, or (2) $x_j[k]$ is independent of $x_j[\alpha]$ but has a closed form, and the iteration is definable, and all control variables are definable at α, then $\tau_{j,\beta}$ is definable.

Proof

(i) Terms preserve definability.Thus, if the assignments all involve either definable variables or composite terms obtained from definable ones, the result is that $\tau_{j,\beta}$ is definable.

(ii) By assumption, τ_{j,β_1} and τ_{j,β_2} are both definable. Now in case (1) we may define $\tau_{j,\beta}$ as τ_{j,β_1} and obtain definability independently of how ill behaved the control variables of φ may be. As for case (2), since the branch taken does affect our result, we define $\tau_{j,\beta}$ as usually and just notice that definability is regained as the control variables are assumed to be definable.

(iii) This case is quite analogous to (ii). In (1) we define $\tau_{j,\beta}$ as the (definable) constant value $x_j[k]$ and in (2) we use the standard cst definition. Definability is regained by our assumptions on the iteration.

■

Theorem 2.5.5 Characterization of Definability for Variables

(i) For an irreducible D-chart D, $\tau_{j,\beta}$ is definable iff the hypothesis (i) of Theorem 2.5.3 or the hypothesis (i) of Theorem 2.5.4 hold.

(ii) For an alternation D, $\tau_{j,\beta}$ is definable iff the hypothesis (ii) of Theorem 2.5.3 or the hypothesis (ii) of Theorem 2.5.4 hold.

(iii) For an iteration D, $\tau_{j,\beta}$ is definable iff the hypothesis (iii) of Theorem 2.5.3 or the hypothesis (iii) of Theorem 2.5.4 hold.

Proof

Theorems 2.5.3 and 2.5.4 prove the "only if" part of this theorem. We shall prove the "if" part by induction on the complexity of D-charts.

Proving (i) is simple, because, given any irreducible elementary D-chart D, it is always true that there exists a term which describes all the assignments made to x_j in D, where we use the symbol x_j to represent $\tau_{j,\alpha}$. So if the assignments depend on $\tau_{j,\alpha}$, and $\tau_{j,\beta}$ is definable, then $\tau_{j,\alpha}$ must be definable. If those assignments do not depend on $x_j[\alpha]$, then the variables occurring in this term have to either be input variables, which are always definable, or definable variables because $\tau_{j,\alpha}$ is so.

As for the proofs of (ii) and (iii), we need to carry out a simultaneous induction.

Say that D is an alternation as in Figure 2.5.2 in which each branch is an irreducible D-chart. We now look at the terms which describe the action of each of the branches on x_j. If they are so that their value is equal and independent of $x[\alpha]$ for all evaluations ι, then the definability of $\tau_{j,\beta}$ forces, as in (i), the desired restrictions on the variables participating in the definition. If the values are still independent of $x[\alpha]$ but they are not equal for all assignments ι, then the definability of $\tau_{j,\beta}$ forces the control variables of φ to be definable at α and imposes the constraint that each term representing the branches be definable as well. If (any) of the values depends on $x[\alpha]$, then the definability of $\tau_{j,\beta}$ forces the definability of $\tau_{j,\alpha}$.

When we have an iteration as in Figure 2.5.3 with irreducible T-branch, then, if $x_j[k]$ is independent of $x_j[\alpha]$ and constant, $\tau_{j,\beta}$ will evaluate to that value, which is, thus, obtained in a definable way. If $x_j[k]$ depends on k but is independent of $x_j[\alpha]$, then the definability of $\tau_{j,\beta}$ forces the existence of a closed form for x_j, the definability of the iteration and that of the control variables at α. The last alternative introduces the further requirement that $\tau_{j,\alpha}$ be definable.

The rest of the proof for alternations and iterations is analogous to the one above but an induction hypothesis is used to deal with branches, instead of using the existence of the term obtainable from straight line code.

■

If $\tau_1, \dots, \tau_1$ are cst's and $\psi(x_1, \dots, x_n)$ a ppf, then $\psi(\tau_1, \dots, \tau_n)$ is a *special program performance formula* (sppf).

Definition 2.5.6

For any program P, if ψ_P is an sppf with no special function symbols f, then P is *definably microanalysable*.

■

We are now at the point where we may characterize those programs whose profile equations can be expressed without the use of any n-place function symbol f.

Theorem 2.5.6

P is definably microanalyzable iff (a) $\psi_D = \psi_P$, (b) for all exit points β in D and any control variable x_j, $\tau_{j\beta}$ is definable, (c) all iterations are definable, and (d) for every control variable and iteration a closed form for the variable exists.

Proof

The "only if" part is proven by inductively constructing the cst's and the ppf which represent P. The "if" part is proven by induction; it uses the assumption $\psi_D = \psi_P$, Theorem 2.5.5 and the observation that the exit points of a construct are the entry points of the subsequent one.

■

The theorems of this section suggest several interesting remarks about the nature of the preservation of definability for different types of objects. We see that proving the preservation of definability for variables is essentially a top-down process. In contrast, proving the preservation of definability for iterations is a bottom-up process, as is the existence of closed forms and the preservation of definability of the iterations. We have also seen that proving the satisfiability of $\psi_D = \psi_P$ may be seen as a bottom-up condition which depends strongly on the nature of the variables appearing in predicates.

Unfortunately, the class of definably microanalyzable programs is rather limited. The sole assumption $\psi_D = \psi_P$ is quite a constraint. In the following chapter we shall see how we may deal with a wider family of programs by changing some of our syntactic constructs. The goal will still be that of obtaining efficiently a representation of program profile equations, but we shall see that the complexity of the evaluation will no longer be linear.

2.6. Summary

In this chapter we have presented our basic approach to the efficient generation of program profiles. A language in which our performance representations of programs can be expressed was introduced in Section 2.1. Section 2.2 presented the semantics of it and the following three sections dealt with issues arising from the representation of programs. It was seen in Theorem 2.4.5 that, to achieve optimally efficient representations, programs need to satisfy conditions on the topology of their D-charts and/or on the behavior of the predicates which govern iterations and alternations. Examples 2.4.1 and 2.4.2 show that in general we cannot expect to achieve much more than what our interpretation function I allows us to achieve.

CHAPTER 3

An Extended Program Performance Language

In Chapter 2 we have seen that a rather restricted class of programs is definably microanalyzable. In fact, Theorem 2.4.5 shows that our problems already begin when we are just considering the efficient generation of the profile equations for a program. Our skeleton procedure presented in Section 1.2 is guaranteed to yield correct profiles for all programs, but its running time may be unacceptable.

The purpose of this chapter is three-fold:

(1) to present alternative methods for finding $\#\varphi$ (all of which are automatizable)

(2) to narrow the gap between what we can do efficiently, in terms of evaluating expressions which represent the profile equations for programs, and our default procedure, and

(3) to extend our model of programs so as to capture more of the features usually found in programming languages.

3.1. On Counting Functions $\#\varphi$

When dealing with an iteration $L = <B, \varphi(\bar{x}), \alpha, \beta>$ with an n-place predicate $\varphi(\bar{x})$, we have introduced the n-place function symbol $\#\varphi$ to denote a function which satisfies the following condition: for any assignment function ι, if $\varphi[\iota]$ is true, then $\#\varphi^{M}[\iota]$ is the number of consecutive times the T-branch of L will be traversed until the flow of control exits the iteration. $\#\varphi$ will be said to be φ's *counting function.*

So far we have not dealt with the issue of finding $\#\varphi$'s. This section will explore this problem from two viewpoints: determining $\#\varphi$ using our knowledge about the counting functions of the (sub)predicates which form φ, and associating the value of $\#\varphi$ with the least nonnegative root of certain special functions determined by the block of the iteration.

It should be clear that, given L as above, $\#\varphi$ depends on both the predicate φ and B. It depends on B because it is there where the transformations to φ's variables are made. This is why $\#\varphi$ and $\#\neg\varphi$ are not directly comparable, as was noted in Section 2.5. Example 3.1.1 illustrates this point.

Throughout this chapter, we shall assume that the control variables are of type numeric, i.e., either of type integer or of type real.

Example 3.1.1

Let us assume that our programming language has a built in definition of the largest possible integer which can be represented, MAXINT. Say that we have an iteration $L = <B, \varphi(j,N), \alpha, \beta>$, in which N is not modified in B, and the only statement affecting j in B is $j := j+1$. Suppose that the predicate φ is $j < N$. Then $\neg\varphi$ is $j \geq N$. Then, $\#\varphi$ may be expressed by $N[\alpha] - j[\alpha]$, and $\#\neg\varphi$ by $\text{MAXINT} - j[\alpha]$.

Thus we see that even though we are only dealing with a complementation operation, the definition of $\#\neg\varphi$ may not be easily found in terms of the definition of φ. We need at least to have information on the environment. In fact, in our example, if the programming language did not have a built in definition of MAXINT, one would not be able to express $\#\neg\varphi$ in it.

■

As was remarked in Section 2.1, by introducing appropriate definitions we may assume that our predicates contain the following six relational operators RO: $<, >, \leq, \geq, =, \neq$. Clearly, they form a set of operators which is closed under negation. So, without loss of generality, we may assume that no predicate φ contains the symbol $\neg$ in its expression.

This will help us deal with the problems suggested by Example 3.1.1. In general, if φ had an occurrence of $\neg$, we would eliminate it using DeMorgan's rules. By appropriate elimination of double negations, one would move the $\neg$ signs next to expressions of the form t_1 RO t_2 , where t_1 and t_2 are terms, and then "absorb" them by using the appropriate relational operator. In fact we shall also assume that all predicates φ will be in disjunctive normal form (with no $\neg$ symbols appearing), i.e., φ will have the form

$$(\varphi_{i_1} \,\&\, \dots \,\&\, \varphi_{i_2}) \text{ OR } (\varphi_{i_2+1} \,\&\, \dots \,\&\, \varphi_{i_3}) \text{ OR } \dots \text{ OR } (\varphi_{i_k+1} \,\&\, \dots \,\&\, \varphi_{i_{k+1}})$$

in which each φ_j is of the form $t_1 \text{RO} t_2$, where t_1, t_2 are terms and RO is one of the six relational operators listed above.

Given a set of functions $f_1, \dots, f_n$, $f = \min\{f_1, \dots, f_n\}$ is defined pointwise, i.e., for every set of values $\mathbf{x}$, $f(\mathbf{x}) = \min\{f_1(\mathbf{x}), \dots, f_n(\mathbf{x})\}$. A similar definition can be given for max, the pointwise maximum of a set of functions.

Given a control variable x and an elementary D-chart D with entrance point α and exit point β, the *action* of D on x is the result $x[\beta]$ as a function of $x[\alpha]$ and of the input values at α for all control variables appearing in D.

Theorem 3.1.1

Given the iteration $L = \langle B, \varphi(\mathbf{x}), \alpha, \beta\rangle$, where $\varphi(\mathbf{x}) = \varphi_1(\mathbf{x}_1) \,\&\, \dots \,\&\, \varphi_n(\mathbf{x}_n)$, if for $1 \leq i \leq n$, $\#\varphi_i$, associated with $\langle B, \varphi_i(\mathbf{x}_i), \alpha_i, \beta_i\rangle$, exists, then

$$\#\varphi = \min\{\#\varphi_1, \dots, \#\varphi_n\}.$$

Proof

We do it by induction on the number of predicates which form φ. If n = 1 there is nothing to prove. So, assume the theorem is true for all predicates

which are a conjunction of n predicates φ_i, and consider $\varphi(\mathbf{x}) = \varphi_1(\mathbf{x}_1) \,\&\, \dots \,\&\, \varphi_n(\mathbf{x}_n) \,\&\, \varphi_{n+1}(\mathbf{x}_{n+1})$. Let ψ represent $\varphi_1(\mathbf{x}_1) \,\&\, \dots \,\&\, \varphi_n(\mathbf{x}_n)$. Then $\varphi = \psi \,\&\, \varphi_{n+1}$.

For any assignment function ι, $\varphi[\iota]$ is true iff $\psi[\iota]$ and $\varphi_{n+1}[\iota]$ are true. Thus, in $L = \langle B, \varphi(\mathbf{x}), \alpha, \beta\rangle$, the first (iteration index) k for which $\varphi[\mathbf{x}[k]]$ is false, is exactly the same k for which either $\psi[\mathbf{x}[k]]$ or $\varphi_{n+1}[\mathbf{x}[k]]$ is false for the first time.

But then, the number of consecutive times that φ will evaluate to true is exactly the number of consecutive times that both ψ and φ_{n+1} will simultaneously evaluate to true, because the body B of the original iteration is the same in all the iterations considered.

Thus $\#\varphi = \min\{\#\psi, \#\varphi_{n+1}\}$.

By induction hypothesis, $\#\psi = \min\{\#\varphi_1, \dots, \#\varphi_n\}$; so

$$\#\varphi = \min\{\#\varphi_1, \dots, \#\varphi_n, \#\varphi_{n+1}\}$$

■

From the proof of this theorem we see that all we need to have as hypothesis on the component iterations $\langle B_i, \varphi_i(\mathbf{x}_i), \alpha_i, \beta_i\rangle$ is that the action on $\mathbf{x}_i$ by the corresponding body be the same, under the same input values at α_i, as the action of B on $\mathbf{x}_i$. We thus obtain the stronger corollary:

Corollary 3.1.2

Given the iteration $L = \langle B, \varphi(\mathbf{x}), \alpha, \beta\rangle$, where $\varphi(\mathbf{x}) = \varphi_1(\mathbf{x}_1) \,\&\, \dots \,\&\, \varphi_n(\mathbf{x}_n)$, if for $1 \leq i \leq n$ $\#\varphi_i$, associated with $\langle B_i, \varphi_i(\mathbf{x}_i), \alpha_i, \beta_i\rangle$, exists, and if for $1 \leq i \leq n$, B_i performs the same action on $\mathbf{x}_i$ as B, then

$$\#\varphi = \min\{\#\varphi_1, \dots, \#\varphi_n\}.$$

Proof

By the proof of Theorem 3.1.1.

■

Unfortunately the case of disjunctions is not as straight forward as the one of conjunctions. The problem is that a predicate may change its truth value several times while the T-branch of the iteration is being consecutively traversed. We shall see examples of this behavior in Chapter 4.

Theorem 3.1.3

Given the iteration $L = \langle B, \varphi(\mathcal{x}), \alpha, \beta \rangle$, where $\varphi(\mathcal{x}) = \varphi_1(\mathcal{x}_1)$ OR ... OR $\varphi_n(\mathcal{x}_n)$, if for $1 \le i \le n$ $\#\varphi_i$ associated with $\langle B_i, \varphi_i(\mathcal{x}_i), \alpha_i, \beta_i \rangle$ exists, and if, for $1 \le i \le n$, B_i performs the same action on $\mathcal{x}_i$ as B, then we have two (algorithmic) ways of computing $\#\varphi$:

(1) Let I_1 be the set of indices j such that $\varphi_j(\mathcal{x}_j[0])$ is true. Let $n_1 = \max_{j \in I_1}\{\#\varphi_j(\mathcal{x}_j[0])\}$. Now consider $\varphi(\mathcal{x}[n_1])$. If false, then $\#\varphi(\mathcal{x}[0]) = n_1$. Otherwise, let I_2 be the set of indices j such that $\varphi_j(\mathcal{x}_j[n_1])$ is true. Let $n_2 = \max_{j \in I_2}\{\#\varphi_j(\mathcal{x}_j[n_1])\}$. If $\varphi(\mathcal{x}[n_1+n_2])$ is false, then $\#\varphi(\mathcal{x}[0]) = n_1 + n_2$. In general we may now find $\#\varphi(\mathcal{x}[0])$, using the above procedure, by: $\#\varphi(\mathcal{x}[0]) = \sum_{i=1}^{k} n_i$, where k is the first index for which $I_k = \phi$.

(2) Let i_1 be the least index i for which $\varphi_i(\mathcal{x}_i[0])$ is true. Let $n_1 = \#\varphi_{i_1}(\mathcal{x}_{i_1}[0])$. Now let i_2 be the least index $i > i_1$ (wrapping around n if necessary) for which $\#\varphi_i(\mathcal{x}_i[n_1])$ is true. Let $n_2 = \#\varphi_{i_2}(\mathcal{x}_{i_2}[n_1])$. Continue in this way until we find i_k and n_k such that, for all $1 \le i \le n$, $\varphi_i(\mathcal{x}_i[\sum_{j=1}^{k} n_j])$ is false. Then $\#\varphi(\mathcal{x}[0]) = \sum_{j=1}^{k} n_j$.

Proof

Given any assignment function i, $\varphi[i]$ will be true if at least one of the $\varphi_j[i]$ is true. The above two methods are based on this same fact in two different ways. That both methods work is proven by induction on the number of participating atomic predicates.

For the case n = 1 there is nothing to prove. Assume that the validity of the methods has been established for n and let $\varphi(\mathcal{x}) = \psi(\mathcal{x})$ OR $\varphi_{n+1}(\mathcal{x})$, where ψ is the disjunction of n predicates. Let us also assume that $\varphi(\mathcal{x}[0])$ is true.

Method (1): We evaluate $\psi(\mathcal{x}[0])$ and $\varphi_{n+1}(\mathcal{x}[0])$, and then find, for those which evaluate to true, $\#\psi(\mathcal{x}[0])$ and $\#\varphi_{n+1}(\mathcal{x}[0])$. As $\varphi(\mathcal{x}[0])$ is true, at least one of the two disjuncts must be true. Let n_1 be the maximum number obtained from the above procedure. n_1 came from, say, ψ (the other case is symmetrical): then $\psi(\mathcal{x}[n_1])$ is false. Thus, we determine whether $\#\varphi_{n+1}(\mathcal{x}[n_1])$ is true or not. If false, we are done and, by induction, this method works correctly. Else we find $n_2 = \#\varphi_{n+1}(\mathcal{x}[n_1])$. Now we only need to look at the truth of $\psi(\mathcal{x}[n_1+n_2])$. If false, we are done, else we find $\#\psi(\mathcal{x}[n_1+n_2])$, and so forth. Thus, this method will correctly find $\#\varphi(\mathcal{x}[0])$, because it will only stop when all predicates become false simultaneously, and then it will compute and record the exact number of iterations that it took for this to happen.

Method (2): This is proven almost identically to the above. In fact, the only difference is that we look at the truth of just one predicate, and from then on alternate to the other one as soon as the current one becomes false. There is no need to consider two cases as above.

■

It should be noticed that our procedures halt only because we are assuming halting programs. It is this the hypothesis that allows us to assert that there will be a finite value of k for which $\varphi(\mathcal{x}[k])$ is false.

Which method is more efficient? Method (2) evaluates less predicates than (1) but must update the values of the control variables more often than (1). Their efficiency will therefore be determined by the behavior of the predicates (i.e., it will depend on whether they change truth value often or not) and by the cost of updating the values of the control variables. We also note that Method (2) can never require fewer iterations than Method (1).

Corollary 3.1.4

Given the iteration $L = \langle B, \varphi, \alpha, \beta \rangle$ where $\varphi(\mathbf{x}) = \varphi_1(\mathbf{x}_1)$ OR $\varphi_2(\mathbf{x}_2)$, if for $1 \leq i \leq 2$ $\#\varphi_i$ associated with $\langle B_i, \varphi_i(\mathbf{x}_i), \alpha_i, \beta_i \rangle$ exists, and if for $1 \leq i \leq 2$, B_i performs the same action on $\mathbf{x}_i$ as B, and $M = \max\{\#\varphi_1, \#\varphi_2\}$, then:

(1) $\varphi_1(\mathbf{x}_1[\#\varphi_2])$ false implies $\#\varphi = M$

(2) $\varphi_2(\mathbf{x}_2[\#\varphi_1])$ false implies $\#\varphi = M$

(3) $(\varphi_1(\mathbf{x}_1[M])$ OR $\varphi_2(\mathbf{x}_2[M]))$ false implies $\#\varphi = M$.

Proof

Any of conditions (1) or (2) imply (3). In case (3), we see that the algorithms described in Theorem 3.1.3 yield $\#\varphi = M$.

■

In the case when a predicate may change its truth value, in the context of an iteration, from true to false at most once, then either hypothesis (1) or (2) of Corollary 3.1.4 holds. In Section 3.2 we shall study this kind of predicates in some detail.

We shall now see that, for the purposes of our analysis, we may choose to look at very simple predicates at the cost of adding some complexity to the body of the iteration, but preserving the counting properties which are of interest to us.

Theorem 3.1.5

Given $L = \langle B, t_1 \text{RO} t_2, \alpha, \beta \rangle$, where t_1 and t_2 are terms, and given two variables x_j, x_i which do not appear in t_1, t_2, or B, then $L_1 = \langle B', x_j \text{RO} x_i, \alpha, \beta \rangle$, where B' differs from B only in that the statements $x_j := t_1$ and $x_i := t_2$ have been added after the statements located at the end of L's T-branch and also added just before the entrance point α, has the same counting functions as L does, i.e.,

$$\#(t_1 \text{RO} t_2) = \#(x_j \text{RO} x_i) \ .$$

Proof

Adding $x_j := t_1$ and $x_i := t_2$ as described in the statement of the theorem preserves the truth of $t_1 \text{RO} t_2$ in the following sense: $t_1 \text{RO} t_2(\mathbf{x}[0])$ is true iff $x_j[0] \text{RO} x_i[0]$ is so, because of the code added just before α. Then, because of the code added at the end of L's T-branch we have that $t_1 \text{RO} t_2(\mathbf{x}[k])$ is true iff $x_j[k] \text{RO} x_i[k]$ is true.

■

Now, the assignments $x_j := t_1$ and $x_i := t_2$ must be understood appropriately in that they may represent a very long programming language statement. However, as we noted in Section 2.5, our terms always represent symbolic sequences in operations of our programming language. We are also implicitly assuming that the evaluation of $t_1 \text{RO} t_2$ has no side effects.

Theorems 3.1.1, 3.1.3 and 3.1.5 allow us to concentrate our attention on the simplest possible predicates, namely those of the form $x_i \text{RO} x_j$. If we are able to find their counting functions in the context of an iteration, then we can find the counting functions of predicates based on them.

To conclude this section, we shall analyze the case when an equality predicate $x = y$ occurs in an iteration predicate φ. The analysis of its effect on $\sharp\varphi$ requires arguments different from the ones presented above. There are two basic cases to consider.

(1) B acts on x and y trivially, i.e., there are no modifications to either variable in the body of the loop. Then $x[\alpha]=y[\alpha]$ implies $x[k]=y[k]$ for all k, and so this predicate will always evaluate to true once the T-branch of the iteration is taken. In this case halting must be assured by other atomic predicates in φ and, for the purposes of determining $\sharp\varphi$, in the light of Theorems 3.1.1 and 3.1.3, it can be discarded.

(2) B acts on x and/or y nontrivially. This case presents more difficulties, because, even though B may act differently on each variable, for some initial values the equality may be preserved throughout several consecutive traversals of the T-branch. For example, we may have the assignments $x:=x*x$ and $y:=2*y$ as the only modifications to x and y in B, but, if $x[\alpha]=y[\alpha]=2$, then $x[1]=y[1]=4$. Other examples can be given in which the equality is preserved an arbitrary prespecified number of traversals of the T-branch. These examples are obtained by using polynomials whose values coincide at a predetermined number of consecutive integer coordinates.

Thus, we see that in case the algebraic transformations to x and y in B are not an identity, we will need to evaluate the predicate at the end of each traversal.

3.2. On Traversal Independent Actions of Iterations on Variables

Definition 3.2.1

Given an iteration L = <B, $\varphi(\bar{x})$, α, β>, we call φ *stable* if for any run of the program, after the flow of control has gone through α, φ changes truth value at most once before the flow of control goes through β.

∎

Stable predicates have pleasantly predictable behavior: after they are evaluated with the values of the control variables at α, their truth value will change at most once before we exit the iteration. Thus, once the change of truth value has occurred, one need not evaluate the predicate any more. Stability is a property which depends exclusively on the action of the body of the iteration on the control variables.

Given a real valued function $f^{\mathbb{R}}$, a *root* for it is any element r of its domain for which $f^{\mathbb{R}}(r) = 0$. Counting functions may be defined pointwise by non negative roots of functions in a way we shall make precise below. This connection will help us find definable counting functions, as described in Section 2.5, and will also set some absolute limits as to how much we may accomplish in this respect.

The sign of a real number r, is defined as follows:

$$\text{sign}(r) = \begin{cases} 1 & \text{if } r>0 \\ 0 & \text{if } r=0 \\ -1 & \text{if } r<0 \end{cases}$$

Given a function symbol f and two points r_1 and r_2, we say that f changes sign if $f^{\mathbb{R}}(r_1)$ and $f^{\mathbb{R}}(r_2)$ have distinct signs. For function symbols f which represent continuous real-valued functions, a change of sign implies that a root exists in the closed interval determined by the points used for the evaluation.

From now on we shall omit the distinction between function symbols f and the real valued function $f^{\mathbb{R}}$ which they represent, unless the ambiguity may prove misleading.

We shall now make explicit an assumption about programs which becomes relevant: programs are assumed not to be self-modifying. In fact, this assumption has been implicitly made throughout our discussion of profile equations. We may then assert that, given an iteration $L = \langle B, x\,\mathrm{RO}\,y, \alpha, \beta\rangle$, the action of B on a control variable the first time its T-branch is traversed is a fixed function of the input variables, i.e., any two times the T-branch of L is entered with the same values for the input variables, the action of B on x and y during the first traversal is the same.

Let us assume that the action of B on the control variables x, y is also independent of the order of traversal, i.e., any two times the T-branch of $L = \langle B, x\,\mathrm{RO}\,y, \alpha, \beta\rangle$, is taken, the action of B on x (and y) is described by the same function. We shall denote the action of B on x by f, and the action on y by g.

Under all of these hypotheses, we have that $x[k] = f^k(x[\alpha])$ and $y[k] = g^k(y[\alpha])$, where f^k (g^k) is the function obtained by composing f (respectively g) with itself k times. We define a new function h as follows:

$$h(k) = f^k(x[\alpha]) - g^k(y[\alpha]).$$

The changes of sign of h will help us find $\# x\,\mathrm{RO}\,y$.

Theorem 3.2.1

Let $L = \langle B, x\,\mathrm{RO}\,y, \alpha, \beta\rangle$ be an iteration and suppose that $x[\alpha]\mathrm{RO}\,y[\alpha]$ is true. Assume further that the action of B on x is f and on y is g, and that these actions are independent of the order of traversal. Let

$$M = \min_k \neg(f^k(x[\alpha])\mathrm{RO}\,g^k(y[\alpha])).$$

Then

$$M = \begin{cases} \min_k h(k)=0 & \text{if RO is } \neq \\ \min_k \mathrm{sign}(h(k-1)) \neq \mathrm{sign}(h(k)) & \text{if RO} \in \{\leq,\geq\} \\ \min_k \mathrm{sign}(h(k-1)) \neq \mathrm{sign}(h(k)) & \text{if RO} \in \{\leq,\geq\} \text{ and = never holds} \\ \min_k \mathrm{sign}(h(k-1)) \neq \mathrm{sign}(h(k)) + 1 & \text{if RO} \in \{\leq,\geq\} \text{ and = holds} \end{cases}$$

Proof

We assume that $x[\alpha]\mathrm{RO}\,y[\alpha]$ is true and (because of the assumption that the iteration halts) that there exists a k such that $\neg x[k]\mathrm{RO}\,y[k]$ is true.

(1) Case when RO is $<$: we are looking for the first k such that $x[k] \geq y[k]$ and, for all $j < k$, $x[j] < y[j]$. This value of k is exactly that for which h changes sign, because under the above conditions, for all $j < k$, $h(j) < 0$ and $h(k) \geq 0$. Thus $M = k$. An analogous argument works for the case when RO is $>$.

(2) If RO $\in \{\leq,\geq\}$ and the minimum k for which h(k) changes sign is such that $h(k) \neq 0$, i.e., $\mathrm{sign}(h(k-1))\times\mathrm{sign}(h(k)) = -1$, then the equality clause of these predicates will not hold (in this run) and so we are back to the case of strict inequalities. However, if RO $\in \{\leq,\geq\}$ and the least k for which h(k) changes sign is such that $h(k) = 0$, then the T-branch of L is traversed a (k+1)st time. Thus our formula is also correct for these two cases.

(3) If RO is $\neq$, we have that $x[\alpha] \neq y[\alpha]$, and then the least k for which $x[k] = y[k]$ will be exactly the least k for which $h(k) = 0$. Thus M is exactly equal to this k.

■

We remark that, when RO is $\neq$, our condition is much harder to determine from a syntactic analysis than in all the other cases. As we shall see in Chapter 4, there are many instances where changes in the sign of h can be determined with some ease. However, the existence of an integer root may not be possible to determine just from the syntax of the expressions defining the function h.

In fact there are instances where real solutions may be determined analytically but integer solutions do not exist. Consider the function $\sin(x)$; we may easily determine in what intervals it changes sign, but unless an *a priori* analytic argument is given for the non existence of integer roots, one may be led to search indefinitely for a non existent integer root.

It should be clear that, under the above conditions, $\mu x[\alpha] \mathrm{RO} y[\alpha]$ is the ceiling of the least nonnegative root of h, where the ceiling of any real number r, $\lceil r \rceil$, is the least integer n such that $n \geq r$.

Stable predicates have a very simple characterization in terms of their associated functions h. We recall that a real valued function h is called *monotonically non decreasing* if, for any two values x, y, $x < y$ implies $h(x) \leq h(y)$. Analogously, h is called *monotonically non increasing* if, for any two values x, y, $x < y$ implies $h(x) \geq h(y)$. Finally h is called *monotonic* if it is either monotonically non decreasing or monotonically non increasing. We then have:

Theorem 3.2.2

Let φ be $x \mathrm{RO} y$. Then φ is stable in L iff the associated function h is monotonic.

Proof

Monotonic functions have the property that, once a change of sign occurs, the previous sign will never appear again. If it did, one would have a reversal in the inequalities. Thus, in the case of functions h associated with predicates RO, by Theorem 3.2.1 this condition describes precisely the

stability conditions.

■

Theorem 3.2.3 A Geometric Characterization of an Iteration's Action

Let $L = \langle B, \varphi(\bar{x}), \alpha, \beta \rangle$ be an iteration whose action on the variable x is the function f, for all traversals. Then, the value of x after k consecutive traversals of the T-branch of L is determined by the intersection of the vertical line $y = x[\alpha]$ with the function f^k.

Proof

This is proven by induction on the iteration index k. By hypothesis we obtain $x[1] = f(x[\alpha])$. Assume that the property holds for k consecutive traversals, then we have: $x[k+1] = f(x[k])$, because the action of L on x is f, and so, by induction,

$$x[k+1] = f(f^k(x[\alpha])) = f^{k+1}(x[\alpha]).$$

It takes a moment's thought to realize that the sequence of points $\{f^k(x[\alpha])\}_{k\in\omega}$ corresponds exactly to the y-coordinates of the points in the plane determined by the intersection of the vertical line $y = x[\alpha]$ and the family of functions $\{f^k\}_{k\in\omega}$.

■

3.3. Algorithmically Definable Counting Functions

Using the results of our last two sections, we shall now analyze the case of definable component predicates. We shall begin by studying some absolute limitations on definability imposed by Theorem 3.2.1.

In Section 2.5 we have seen that, in order to find definable $\mu\varphi$'s, we need to have closed form expressions for the value of the kth iteration of the control variables. Moreover, Theorem 3.2.1 tells us that we must also be able to *solve* for k.

In the particular case when the expressions found for the values of the control variables after the kth traversal of the T-branch of an iteration are a polynomial in k, our problem then amounts to knowing whether we may find an (algebraic) expression involving radicals which would yield the first nonnegative root of this polynomial.

The fundamental Theorem of Galois theory [Art71] states that quintics are unsolvable in radicals. However, our knowledge of the existence of a nonnegative root for those polynomials we are interested in may lead us to think that a formula for $\#\varphi$ could exist. Unfortunately this is not the case.

Theorem 3.3.1

If all polynomials with rational coefficients and a nonnegative root were solvable, then all polynomials would be solvable.

Proof

Let $p(x)$ be a polynomial with rational coefficients. Let r be its real root with the largest absolute value, and K be a rational number larger than the absolute value of r. Then $p_1(x)=p(x-K)$ is also a polynomial with rational coefficients all of whose real roots are nonnegative. But then by hypothesis one would be able to find a splitting field F for p_1. However, as K belongs to the base field, F would also be a splitting field for p, given that $p_1(s) = 0$ iff $p(s-K) = 0$, and $s \in F$ implies that $s-K \in F$.

■

We shall now see two examples which illustrate the absolute algebraic limitations on the definability of counting functions. The first is a generic example exhibiting the limitations imposed by the unsolvability of higher order equations, and the second illustrates a different problem, that of exponential equations.

Example 3.3.1

Consider $L = \langle B, x \,\mathrm{RO}\, n, \alpha, \beta\rangle$, where in B the only assignment which modifies x is $x:=i_1*i_2*i_3*i_4*i_5$, and where each of the variables i_j is modified once in B by the statement $i_j:=i_j+\tau_j$, where no τ_j is modified in B, and n is also not modified in B.

With th~~-~~se conditions it is clear that

$$x[k] = (i_1[0]+k*\tau_1)*(i_2[0]+k*\tau_2)*(i_3[0]+k*\tau_3)*(i_4[0]+k*\tau_4)*(i_5[0]+k*\tau_5)$$

and the test performed to see if the T-branch is to be traversed again is

$$x[k] \;\mathrm{RO}\; n[0].$$

A sufficient condition for L to be a halting iteration is that all τ_i's be positive. But then, as $x[k]$ is a quintic in k which can be transformed, by giving appropriate values to the τ_j's and $i_j[\alpha]$'s, into a particular non solvable quintic: we cannot expect L to be definable.

■

Example 3.3.1 is quite *ad hoc*. Rarely transformations made to control variables are of such algebraic complexity [Knu71, Hen80]. Our next example shows that, even when we assume very simple conditions, definability may not be achievable.

Example 3.3.2

Consider $L = \langle B, x<y, \alpha, \beta\rangle$ and say that the action of B on x is $a*x+b$, the action on y is $c*y+d$, and that a, b, c, d are not changed by B. After traversing k times the T-branch of L, one has (assuming $a \neq 1 \neq c$)

$$h(k) = \left(a^k x[\alpha] + b\left(\frac{a^k-1}{a-1}\right)\right) - \left(c^k y[\beta] + d\left(\frac{c^k-1}{c-1}\right)\right).$$

This is an exponential equation on the iteration index k which does not have a symbolic expression for its roots, even when using logarithms, unless a = c . Thus L is not definable even though it has a very simple predicate and simple actions on its control variables.

∎

The following theorem gives us one instance when definability for an iteration with a composite predicate can be obtained from the definability of some iterations with atomic predicates.

Theorem 3.3.2

Given the iteration $L = \langle B, \varphi(\bar{x}), \alpha, \beta \rangle$, where $\varphi(\bar{x}) = \varphi_1(\bar{x}_1)$ & ... & $\varphi_n(\bar{x}_n)$, if for $1 \leq i \leq n$ each $\langle B_i, \varphi_i(\bar{x}_i), \alpha_i, \beta_i \rangle$ is definable and B_i acts on $\bar{x}_i$ in the same way as B does, then the evaluation cost of $\#\varphi = \min\{\hat{\varphi}_1, \ldots, \hat{\varphi}_n\}$ is (still) linear on the length of the iteration's description.

Proof

By the assumption of definability, the evaluation of each $\hat{\varphi}_i$ is done in linear time. Then, evaluating $\#\varphi$ takes at most n times the longest evaluation time for an individual $\hat{\varphi}_i$.

∎

Thus we see that, even though we need to apply an algorithm to obtain $\#\varphi$, we can still achieve the goal of linear evaluation time. However, this is not the case for disjunctions.

Theorem 3.3.3

Given the iteration $L = \langle B, \varphi(\bar{x}), \alpha, \beta \rangle$, where $\varphi(\bar{x}) = \varphi_1(\bar{x}_1)$ OR ... OR $\varphi_n(\bar{x}_n)$, if for $1 \leq i \leq n$ each $\langle B_i, \varphi_i(\bar{x}_i), \alpha_i, \beta_i \rangle$ is definable and B_i acts on $\bar{x}_i$ in the same way as B does, then the evaluation cost of $\#\varphi$ is proportional to the number of iterations of the algorithms described in Theorem 3.1.3.

Proof

In either method described in Theorem 3.1.3 we may bound the cost incurred at each iteration by looking at the cost of evaluating each individual

$\hat{\varphi}_i$. As these last costs are constant, we obtain the desired bound.

∎

In the last two cases, $\#\varphi$ may be called *algorithmically definable*. We observe that, as in the case of disjunctions one does not have an a priori upper bound on the number of cycles that the algorithm will perform, we no longer have a linear time evaluation cost. However, different empirical studies [Hen80, Woo79, Knu71] have consistently shown that long predicates hardly ever occur in practice. When φ is solely composed of stable atomic predicates, the number of them in φ bounds the number of times the algorithm will cycle.

3.4. A Solution to the Limitations of Lemmas 2.4.3 and 2.4.4

In our Examples 2.4.1 and 2.4.2 we showed that even structurally simple programs can arbitrarily misbehave, thereby making the successive evaluation of their predicates the only way to obtain correct profiles. It is also true that the hypotheses of Lemmas 2.4.3 and 2.4.4 are quite strong and they may be difficult to check automatically. (The automatic verification of these and other hypotheses will be explored in Chapter 7.)

We thus need ways to approach this problem more efficiently than by the skeleton procedure. As our examples show that the evaluation of inner predicates may be unavoidable, we shall not try to do better than that, but to do it as rapidly as possible. For this purpose we shall introduce a new three-place special symbol.

The new three-place special symbol to be added to our program performance language is FORTODOOD. The inductive rule of usage to obtain *algorithmic ppf*'s is the following:

if f is an n-place function symbol, ψ an algorithmic ppf, and k a symbol not appearing in ψ, then

$$\text{FORTODOOD}(k=1, f, \psi)$$

is an algorithmic ppf.

As done with our other special symbol, we shall always write the above in the form

$$\text{FOR } k=1 \text{ TO } f \text{ DO } \psi \text{ OD} \ .$$

The interpretation of this new symbol will be defined after we associate algorithmic ppf's with D-charts. Otherwise the definition of the interpretation function becomes unnecessarily complex.

This new special symbol will only be used when we encounter certain patterns in the D-charts, namely, alternations within iterations or iterations within iterations.

The Case of Alternations Within Iterations

Assume we have a situation like that in Figure 3.4.1, where D_1 and D_2 are elementary D-charts represented by the ppf's ψ_{D_1} and ψ_{D_2} respectively, and where ψ_1 is the ppf which represents B_1, ψ_2 the one for B_2. The ppf associated with this elementary D-chart is

$$\text{IF } \varphi_0, \#\varphi_0 \text{ THEN } \psi_{D_1} \text{ IF } \varphi_k, 1 \text{ THEN } \psi_1 \text{ ELSE } \psi_2 \text{ FI } \psi_{D_2} \text{ ELSE } \Lambda \text{ FI} \ .$$

The algorithmic ppf associated with this elementary D-chart, where we have taken care of the alternation with predicate φ_k which was within the iteration with predicate φ_0, is:

$$\text{IF } \varphi_0, \#\varphi_0 \text{ THEN } \psi_{D_1}\psi_{D_2} \text{ ELSE } \Lambda \text{ FI FOR } k=1 \text{ TO } \#\varphi_0 \text{ DO IF } \varphi_k, 1 \text{ THEN } \psi_1 \text{ ELSE } \psi_2 \text{ FI OD}$$

In fact the transformation "pull out of the iteration" should be applied once simultaneously to all those alternations within the iteration with predi-

cate φ_0 which do not satisfy the hypothesis of Lemma 2.4.3. It should be clear that an inductive definition using the above criterion will enable us to handle all instances of alternations within iterations.

The Case of Iterations Within Iterations

Assume we have a situation like that in Figure 3.4.2, where D_1 and D_2 are elementary D-charts represented by the ppf's ψ_{D_1} and ψ_{D_2} respectively, and where ψ is the ppf which represents B. The ppf associated with this elementary D-chart is

$$\text{IF } \varphi_0, \#\varphi_0 \text{ THEN } \psi_{D_1} \text{ IF } \varphi_k, \#\varphi_k \text{ THEN } \psi \text{ ELSE } \Lambda \text{ FI } \psi_{D_2} \text{ ELSE } \Lambda \text{ FI} \ .$$

The algorithmic ppf associated with this elementary D-chart, where we have

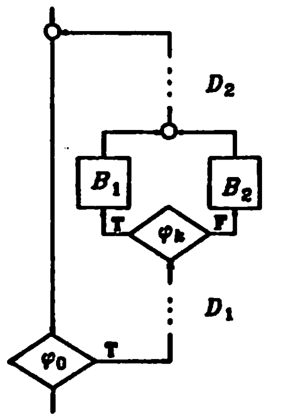

Figure 3.4.1

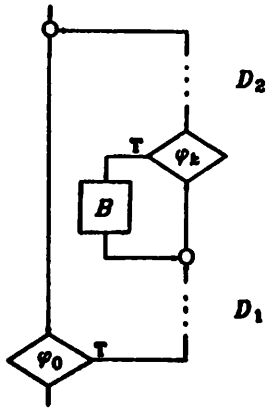

Figure 3.4.2

taken care of the iteration with predicate φ_b which was within the iteration with predicate φ_0, is:

IF φ_0. $\#\varphi_0$ THEN $\psi_{D_1}\psi_{D_0}$ ELSE Λ FI FOR $k=1$ TO $\#\varphi_0$ DO IF φ_b. $\#\varphi_b$ THEN ψ ELSE Λ FI OD

As in the case of alternations this transformation should be applied once simultaneously to all those iterations within the iteration with predicate φ_0 which do not satisfy the hypothesis of Lemma 2.4.4. Again, it should be clear that an inductive definition using this criterion will enable us to handle all instances of iterations within iterations.

We extend our interpretation function I to handle algorithmic ppf's. Its definition, however, will be algorithmic in nature, and only definability assumptions may allow faster evaluation time than that of running the skeleton.

Interpretation of Algorithmic ppf's

(1) Case of an alternation with irreducible branches and predicate $\varphi_1(\bar{x}_1)$ within an iteration with predicate $\varphi_0(\bar{x}_0)$: given an assignment function $\iota:V \to \mathbb{R}$,

$$I(\text{FOR } k=1 \text{ TO } \#\varphi_0 \text{ DO IF } \varphi_1.\ 1 \text{ THEN } B_1 \text{ ELSE } B_2 \text{ FI OD})[\iota]$$

is the following procedure: for k equal to 1 up to $\#\varphi_0[\iota]$, evaluate $\varphi_1(\bar{x}_1[k-1])$, keep appropriate counters C_1, C_2 for the T-branch and the F-branch respectively, and update the values of the control variables. When done, the final counters C_1 and C_2 must satisfy $C_1 + C_2 = \#\varphi_0[\iota]$. The output of I is the string $C_1{}^{\mathbb{R}}B_1C_2{}^{\mathbb{R}}B_2$.

(2) Case of an iteration with an irreducible T-branch B_1, and predicate $\varphi_1(\bar{x}_1)$, within an iteration with predicate $\varphi_0(\bar{x}_0)$: given an assignment function $\iota:V \to \mathbb{R}$

$$I(\text{FOR } k=1 \text{ TO } \#\varphi_0 \text{ DO IF } \varphi_1.\ \#\varphi_1 \text{ THEN } B_1 \text{ ELSE } \Lambda \text{ FI OD})[\iota]$$

is the following procedure: for k equal to one up to $\#\varphi_0[\iota]$, evaluate $\varphi_1(\bar{x}_1[k-1])$, update the values of the control variables, and keep a sum of the values in the counter C_1. When done, C_1 must be equal to $\sum_{k=1}^{\#\varphi_0[\iota]}$ $\#\varphi_1(\bar{x}_1[k-1])$. The output of I is the string $C_1{}^{\mathbb{R}}B_1$.

(3) Other cases. They are analogous to the above two. The only difference is that the counters C_1 and C_2 now apply to subformulae and not to basic blocks. C_1 and C_2 will be profile coefficients multiplying the evaluation of the corresponding (sub)formulae.

Theorem 3.4.1

Algorithmic program performance formulae always represent the profile equations.

Proof

By a simple induction on the complexity of algorithmic program performance formulae, noting that the "algorithmic" special symbol is only introduced when the original ppf does not represent the profile equations.

■

Example 3.4.1

We shall now show how we are able to deal with our Example 2.4.1. (see Figure 2.4.3). In this case one applies case (1) of the rules for interpreting algorithmic ppf's. The algorithmic ppf for this case is

$$\text{FOR } k=1 \text{ TO } N \text{ DO IF } A[k] \geq 0.\ 1 \text{ THEN } B_1 \text{ ELSE } \Lambda \text{ FI OD}$$

where B_1 is the basic block corresponding to the alternation's T-branch in which the sum S is performed.

The interpretation is as follows: for k equal to 1 up to N, evaluate the predicate $A[k] \geq 0$. If true, record this by incrementing C_1. If false, do nothing (because the ELSE branch is Λ). When finished, output $C_1^R B_1$.

∎

Introducing the new special symbol FORTODOOD has certainly solved our problem of finding symbolic representations for the profile equations of D-charts, but the drawback is that we have ended up with an algorithmic definition for L whose evaluation time may be as bad as that of running the skeleton.

We present two cases where the expected cost of evaluating the algorithmic ppf is smaller than that of running the skeleton:

(1) When we are given a definable iteration L with predicate $\varphi_0(\mathbf{x}_0)$, an alternation within this iteration with predicate $\varphi_1(\mathbf{x}_1)$, and closed form expressions for φ_1's control variables $\mathbf{x}_1$.

(2) When we are given a definable iteration L with predicate $\varphi_0(\mathbf{x}_0)$ and another definable iteration $\varphi_1(\mathbf{x}_1)$ within it for which we have closed form expressions for its control variables $\mathbf{x}_1$.

In both of these cases, when evaluating the algorithmic ppf, we expect to outperform the running time of the skeleton because the truth of predicates need not be established each time a T-branch is to be taken. The fact that we have closed forms for the control variables of the inner predicate allows us to "update" them in one (possibly complex) evaluation. The skeleton will be updating them each time the corresponding branch is traversed. If the iterations are to be traversed a substantial number of times, then running the skeleton should take much longer than evaluating the algorithmic ppf.

3.5. Analyzing a Larger Class of Programs

We have adopted, for the purposes of our study, the representation of programs based on D-charts. From the work of Böhm and Jacopini [Böh66], we know that any computational flowchart can be converted, perhaps at the expense of adding some extra boolean variables, into a D-chart which will preserve the functionality of the original program. Moreover, it is also known [Led75] that one needs to introduce auxiliary boolean variables only when "untangling" a loop which has more than one exit. It is the presence of blocks which are not one-in one-out which forces the introduction of auxiliary variables.

Most modern programming languages have a richer set of control structures than those naturally represented by our program performance formulae, which satisfy the property of single-entry single-exit points. We shall now see how to deal with them. Once we exhibit the appropriate reductions to our formulae, all our definability results will apply.

In particular, we shall deal with the so-called D'-charts [Led75], which are D-charts with two additional rules of formation. They are depicted in Figures 3.5.1 and 3.5.2.

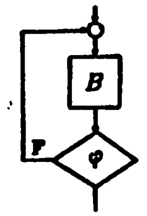

Figure 3.5.1

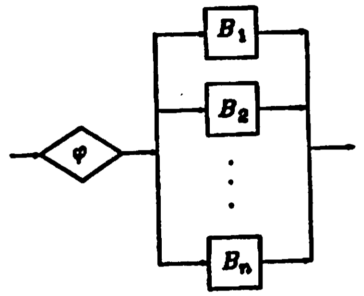

Figure 3.5.2

Figure 3.5.1 is called a *repeat-until* iteration, and Figure 3.5.2 is a *case statement* for $1 \leq i \leq n$. In Figures 3.5.3 and 3.5.4 we give their equivalent representations in terms of our D-charts.

We may then treat these control structures using the depicted equivalences. We may perform similar reductions with any construct reducible to a D-chart.

So far our methods have not dealt with control variables whose type is not numeric. There is a good reason for it, and it is because the behavior characteristics of those programs are very difficult to determine from their

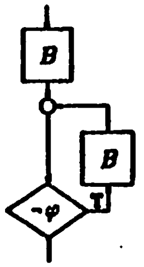

Figure 3.5.3

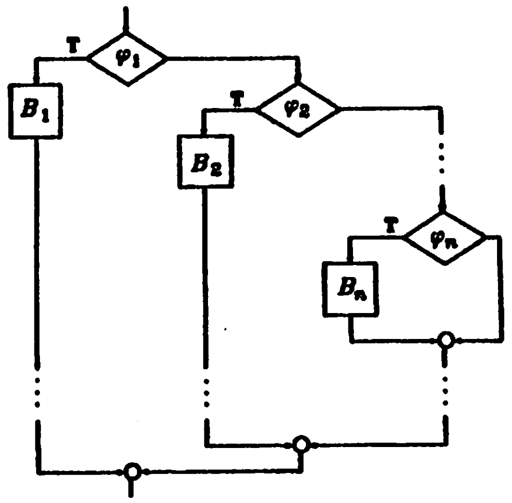

Figure 3.5.4

syntax. In Chapter 7 we shall present some examples of these programs.

3.6. Some Issues Concerning Unrestricted GO-TO's

The controversy on the usage of unrestricted GO-TO's in programming is a subject which has a long history. There have been dozens of authors contributing their ideas to the subject. We shall not present an in-depth analysis of the pros and cons of unrestricted GO-TO's but just make some remarks as to how their appearance in programs affects our methods and goals. In [Knu74, Dij72, Led75] the interested reader may get acquainted with some of the issues and arguments in this controversy.

For us, the basic problem with unrestricted GO-TO's is that they destroy the one-in one-out flow of control property of D-charts. This is troublesome when determining the definability of a variable. With the one-in one-out property, one can naturally order the basic blocks so that a basic block B is not processed until all of its predecessors B_i are processed; i.e., the nodes B_i are always traversed before B in any run.

When the one-in one-out property is not satisfied, as in Figure 3.6.1, one can no longer do this. The immediate problem is that one needs several passes through the graph to determine the existing dependencies. This increases the running time needed to find the performance representation of the program [All76]. In Figure 3.6.1 we may also see that B_3 is both a predecessor and a successor of B_2, because there are runs which traverse B_3 before B_2 and others which traverse B_2 before B_3.

A different aspect of this same problem is that finding definable iterations (regardless of the fact that iterations are semantically different from those in D-charts, in that they may have several entry and exit points) becomes much harder. In Figure 3.6.1, we see that B_2 is also part of the

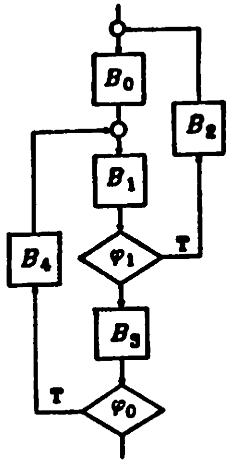

Figure 3.6.1

iteration which has φ_1 as the governing predicate. If φ_1's control variables are modified in the iteration with predicate φ_0, then the mere existence of a closed form for φ_1's control variables is already a difficult problem. In fact, a closed form for them can never exist as a pure algebraic expression, because of the dependence on φ_0's truth value.

Thus, even though our methods may handle unrestricted GO-TO's, the cost of doing it is normally felt at all levels. Building a performance representation will take longer, because multiple passes through the code will be needed. Iterations, which now have to be defined as cycles in the directed graph representing the flow of control of the program, will be less likely to be definable. It then becomes more probable that fewer iterations

in a program will be "linearizable" and so the evaluation cost of our performance representation will tend to be close to that of running the skeleton.

3.7. Summary

In this chapter we have dealt with several shortcomings of our program performance language introduced in Chapter 2. We have first studied how to obtain the (crucial) counting functions $\#\varphi$ for nonatomic predicates φ. We have then analyzed a special kind of predicates, the stable ones, which help us evaluate $\#\varphi$ in an efficient manner. Stability was also characterized in terms of the behavior of a real valued function which can be determined from the text of the program.

We have then introduced a new 3-place special symbol which allows us to represent the profile equations of arbitrary D-charts. The interpretation function I was expanded accordingly, but now, to deal correctly with this new symbol, one has to define I in an algorithmic form. Which method, the algorithmic ppf or the skeleton, will have a shorter execution time depends on the definability hypotheses satisfied in a given case. Finally, we have noted how the violation of the one-in one-out property of D-charts by arbitrary GO-TO's makes our analysis more costly.

CHAPTER 4

The Linear Function Case

In several empirical studies of programs [Knu71b, Woo79, Hen80], it has been observed that most modifications made to control variables in the body of iterations are, algebraically, very simple. This has motivated our detailed study of the counting functions of iterations in which the control variables are modified according to linear functions. We shall see that under these hypotheses one can efficiently obtain, always at run time, sometimes at compile time, *all* the characteristics of the counting functions.

Throughout this chapter we shall consider exclusively an iteration $L = \langle B, x\,\mathrm{R0}\,y, \alpha, \beta\rangle$, where the actions of B on x and y are linear functions. We assume the action of B on x to be $f(x) = ax + b$, and the action on y to be $g(y) = cy + d$, where a, b, c and d are names of variables whose value does not change in B.

Under these hypotheses the closed forms for $x[k]$ and $y[k]$ are (recall Example 3.3.2):

$$x[k] = \begin{cases} x[\alpha] + kb & \text{if } a = 1 \\ a^k x[\alpha] + b\left(\frac{a^k - 1}{a - 1}\right) & \text{if } a \neq 1 \end{cases}$$

$$y[k] = \begin{cases} y[\alpha] + kd & \text{if } c = 1 \\ c^k x[\alpha] + d\left(\frac{c^k - 1}{c - 1}\right) & \text{if } c \neq 1 \end{cases}$$

We shall use the function $h(k) = x[k] - y[k]$ to analyze the behavior of $x\,\mathrm{R0}\,y$ in L. However, it proves useful to extend the definition of h to all real numbers r, even though, from the viewpoint of $\#x\,\mathrm{R0}\,y$, the function h is

interesting only at integer points. The following section analyzes some useful properties of certain special families of linear functions. For any real number t and function f, we define $f^0(t) = t$.

4.1. Some Properties of Special Families of Linear Functions

Consider the real valued linear function $f(t) = at + b$, where a and b are two real numbers. The composition of f with itself k times yields

$$f^k(t) = \begin{cases} t + kb & \text{if } a = 1 \\ a^k t + b\left(\frac{a^k - 1}{a - 1}\right) & \text{if } a \neq 1 \end{cases}$$

Theorem 3.2.3 tells us that the family $\{f^n\}_{n \geq 0}$ deserves study. When f is a linear function, it turns out that $\{f^n\}_{n \geq 0}$ possesses one characteristic which accounts for most of the family's good behavior: either all the straight lines which represent the elements of the family are parallel or they meet at a single point.

Theorem 4.1.1

Let f be the linear function $f(t) = at + b$. If $a = 1$, then the functions f^n in the family $\{f^n\}_{n \geq 0}$, represent parallel lines in the plane. Otherwise, the point $\left(\frac{b}{1-a}, \frac{b}{1-a}\right)$ belongs to each member of $\{f^n\}_{n \geq 0}$.

Proof

We have given above the expressions for f^k. From them, one can see that, if $a = 1$, $f^k(t) = t + kb$, and, for $k_1 \neq k_2$, $|f^{k_1}(t) - f^{k_2}(t)| = |k_1 - k_2|b$, which is independent of t. Thus, the two straight lines are parallel.

When $a \neq 1$ we have

$$
\begin{aligned}
f^k\left(\frac{b}{1-a}\right) &= a^k \frac{b}{1-a} + b\frac{a^k-1}{a-1} \\
&= \frac{1}{1-a}(ba^k - b(a^k-1)) \\
&= \frac{1}{1-a}(ba^k - ba^k + b) \\
&= \frac{b}{1-a}\ .
\end{aligned}
$$

So, the point $\left(\frac{b}{1-a}, \frac{b}{1-a}\right)$ belongs to each of the straight lines represented by functions in $\{f^n\}_{n\geq 0}$.

∎

The point $\left(\frac{b}{1-a}, \frac{b}{1-a}\right)$ will be called the ***intersection point*** of $\{f^n\}_{n\geq 0}$, and plays an important role when determining halting conditions. When f is linear, the family $\{f^n\}_{n\geq 0}$ will be called a ***special family*** of functions.

The value of the coefficient a determines all the essential characteristics of the behavior of the family $\{f^n\}_{n\geq 0}$, in particular, when we consider the intersection of $\{f^n\}_{n\geq 0}$ with the vertical line $y = x[\alpha]$. There are only seven cases to consider.

Let $A = x[\alpha] + \frac{b}{a-1}$.

(1) $a > 1$

$$
\begin{aligned}
\lim_{k\to\infty} f^k(x[\alpha]) &= \lim_{k\to\infty} a^k(A) + \frac{b}{1-a} \\
&= \begin{cases} \text{sign}(A)\infty & \text{if } x[\alpha] \neq \frac{b}{1-a} \\ \frac{b}{1-a} & \text{otherwise} \end{cases}
\end{aligned}
$$

(2) $a = 1$

$$\lim_{k\to\infty} f^k(x[\alpha]) = \lim_{k\to\infty} (x[\alpha] + kb) = \text{sign}(b)\infty$$

(3) $0 < a < 1$

$$
\begin{aligned}
\lim_{k\to\infty} f^k(x[\alpha]) &= \lim_{k\to\infty} a^k(A) + \frac{b}{1-a} \\
&= \frac{b}{1-a}\ .
\end{aligned}
$$

(4) $a = 0$

$$\lim_{k\to\infty} f^k(x[\alpha]) = b\ .$$

For two of the cases when a is negative, it is convenient to introduce two auxiliary functions: f_m and f_p. They give us bounds on the values that the family $\{f^n\}_{n\geq 0}$ takes. Consider a given value $x[\alpha]$. We define our new functions by:

$$
\begin{aligned}
f_m &= -|a|^k\left(x[\alpha] + \frac{b}{a-1}\right) + \frac{b}{1-a} \\
f_p &= |a|^k\left(x[\alpha] + \frac{b}{a-1}\right) + \frac{b}{1-a}
\end{aligned}
$$

(index m stands for *minus* and the p for *plus*).

When $a < 0$ and $a \neq -1$, we can easily see that $f^{2k}(x[\alpha]) = f_p(2k)$ and $f^{2k+1}(x[\alpha]) = f_m(2k+1)$. So the values of $\{f^n\}_{n\geq 0}$, when evaluated at the point $x[\alpha]$, oscillate between those of f_m and f_p. We must also notice that, under these hypotheses, f_m and f_p are monotonic functions. This is most clearly seen when one observes that the only change in the values of f_m and f_p is given by the change of value in $|a|^k$. As the real valued exponential functions are monotonic, f_m and f_p are monotonic.

Table 4.1.1 summarizes the monotonicity properties of f_m and f_p.

function	assumptions	behavior
f_m	$A \geq 0$ & $\lvert a\rvert > 1$	monotonically nonincreasing
f_m	$A \geq 0$ & $\lvert a\rvert < 1$	monotonically nondecreasing
f_m	$A \leq 0$ & $\lvert a\rvert > 1$	monotonically nondecreasing
f_m	$A \leq 0$ & $\lvert a\rvert < 1$	monotonically nonincreasing
f_p	$A \geq 0$ & $\lvert a\rvert > 1$	monotonically nondecreasing
f_p	$A \geq 0$ & $\lvert a\rvert < 1$	monotonically nonincreasing
f_p	$A \leq 0$ & $\lvert a\rvert > 1$	monotonically nonincreasing
f_p	$A \leq 0$ & $\lvert a\rvert < 1$	monotonically nondecreasing

Table 4.1.1 Behavior of f_m and f_p.

There is another property of these functions which is of interest to us. In the semiplane $\{<x,y> : x \geq 1\}$, they are symmetrical (i.e., mirror images) with respect to the horizontal line $x = \frac{b}{1-a}$. A way to see this is to perform a displacement along the vertical axis of $\frac{b}{1-a}$ units. This transforms $f_m(k)$ into $f'_m(k) = -|a|^k A$ and $f_p(k)$ into $f'_p(k) = |a|^k A$. Thus $f'_m(k) = -f'_p(k)$. Clearly then, f_m and f_p are symmetrical with respect to the horizontal line $x = \frac{b}{1-a}$.

(5) $-1 < a < 0$

$$\lim_{k\to\infty} f_p(k) = \lim_{k\to\infty} |a|^k A + \frac{b}{1-a} = \frac{b}{1-a}\ .$$

$$\lim_{k\to\infty} f_m(k) = \lim_{k\to\infty} -|a|^k A + \frac{b}{1-a} = \frac{b}{1-a}\ .$$

In this case, the values $\{f^n(x[\alpha])\}_{n\geq 0}$ form a sequence which converges to $\frac{b}{1-a}$. Moreover, in the plane, all even indexed elements of this sequence are on the same side of the line $x = \frac{b}{1-a}$. On the other side of this line are all of the odd indexed elements. Thus the values of the sequence cross over the line $x = \frac{b}{1-a}$ every time. This says that h will change sign an infinite number of times iff $b = 0$.

(6) $a = -1$

This case is quite peculiar, in that the whole family $\{f^n\}_{n\geq 0}$ reduces to two functions: $f(t) = -t + b$ and $f^2(t) = t$. Thus, for all $k \geq 0$, $f^{2k}(x[\alpha]) = x[\alpha]$ and $f^{2k+1}(x[\alpha]) = -x[\alpha] + b$.

(7) $a < -1$

$$\lim_{k\to\infty} f_p(k) = \begin{cases} \text{sign}(A)\infty & \text{if } x[\alpha] \neq \frac{b}{1-a} \\ \frac{b}{1-a} & \text{otherwise} \end{cases}$$

$$\lim_{k\to\infty} f_m(k) = \begin{cases} -\text{sign}(A)\infty & \text{if } x[\alpha] \neq \frac{b}{1-a} \\ \frac{b}{1-a} & \text{otherwise} \end{cases}$$

The sequence $\{f^n(x[\alpha])\}_{n\geq 0}$ diverges. Moreover, the absolute value of the elements increases monotonically. The values of the odd indexed elements will diverge with signs different from those with even indices. As in (5), they will also be on different sides of the line $x = \frac{b}{1-a}$.

Figures 4.1.1 through 4.1.6 depict typical situations of $\{f^n\}_{n\geq 0}$ for each case. We have omitted the case for a = 1, where parallel lines making a forty five degree angle with the horizontal axis and d units apart from each other would be obtained.

4.2. A Real Valued Function Approach

In Section 3.2 the function $h(k) = x[k] - y[k]$ was introduced and used to characterize the stability properties of an iteration predicate. Moreover, Theorem 3.2.1 gave us a relationship between the least nonnegative root of h and the counting function of the iteration.

When analyzing an iteration, the values $x[\alpha]$ and $y[\alpha]$ can be viewed as parameters. In fact, once we fix them, h becomes a function of the iteration index k only. For the purposes of finding the roots of h, it is better to consider h to be defined for all real values. When using the hypotheses of this chapter, if a and c are nonnegative, viewing h as a real valued function presents no problem. In the other cases we shall see that using f_p, f_m, g_p and g_m, where g_p and g_m are the equivalents of f_p, f_m for y, will allow us to perform our analysis.

Under the hypotheses of this chapter, the analysis of the counting functions for iterations is reduced to the study of only four kinds of functions h. We only need to find the nonnegative roots for the following four types of functions:

I $\quad h(r) = rA + B - (rC + D)$

II $\quad h(r) = a^r A + B - D$

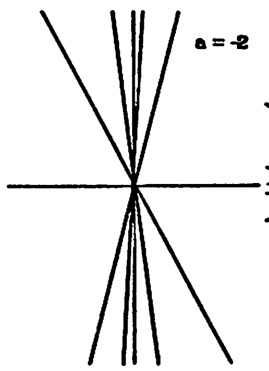

Figure 4.1.1

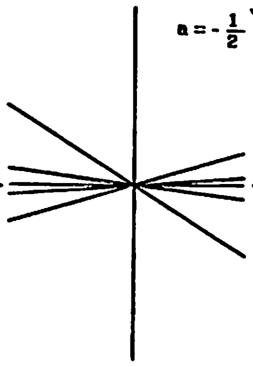

Figure 4.1.2

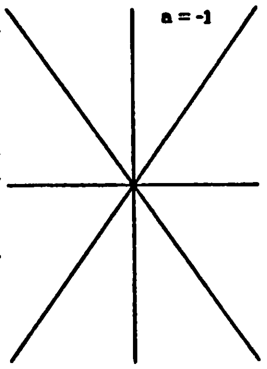

Figure 4.1.3

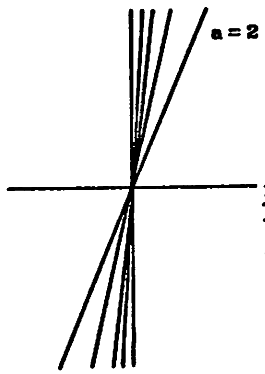

Figure 4.1.4

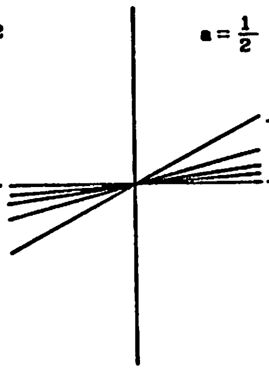

Figure 4.1.5

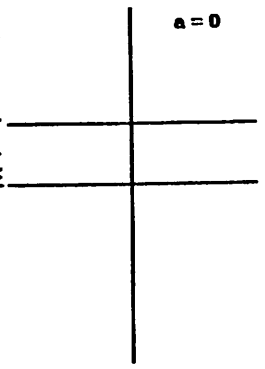

Figure 4.1.6

III $h(r) = a^r A + B - (rC + D)$

IV $h(r) = a^r A + B - (b^r C + D)$

where a, A, B, C and D are assumed constant, A, C non zero, and a positive.

Functions of Type I

$$h(r) = 0 \text{ iff } rA + B = rC + D \text{ iff } r(A - C) = D - B$$

In order to find the root, we must divide by (A - C), thus we must have that A $\neq$ C, i.e., the two straight lines must not be parallel. Thus:

root existence condition: $A \neq C$

root expression: $r = \frac{D - B}{A - C}$

Functions of Type II

$$h(r) = 0 \text{ iff } a^r A + B = D \text{ iff } a^r = \frac{D - B}{A}$$

As a^r is always positive, we must have that $\text{sign}(\frac{D - B}{A}) = 1$, i.e., that the exponential function $y = a^r A + B$ crosses the horizontal line $x = D$. Thus,

root existence condition: $\text{sign}\left(\frac{D - B}{A}\right) = 1$.

root expression: $r = \frac{\log\left(\frac{D - B}{A}\right)}{\log(a)}$

Functions of Type III

$$h(r) = 0 \text{ iff } a^r A - rC + (B - D) = 0.$$

In this case we will have nonnegative roots iff the straight line determined by $y = rC + D$, and the exponential function determined by $y = a^r A + B$, meet in the positive semiplane. The necessary analytic conditions for this to happen are not as elegant as the ones above.

The *tangent slope* is that value of the slope of $y = rC + D$, i.e., that value of $\bar{C}$, which makes the exponential function and the straight line meet in exactly one point. It is determined by the unique solution r of the following equation:

$$Aa^r + B = \ln(a)Aa^r r + D .$$

There is no general way of expressing the root r of this equation in an algebraic way. Hence, r needs to be found using some numerical method. Thus,

root existence conditions: see Table 4.2.1

root expression: does not exist (in general)

Functions of Type IV

$$h(r) = 0 \text{ iff } a^r A - b^r C + (B - D) = 0 .$$

In this case we will have nonnegative roots iff both exponentials meet. In any such case, we will only have one root. Thus,

root existence conditions: see Table 4.2.2

root expression: does not exist (in general)

conditions	property
$A > 0$ & $D > A + B$ & any C	one root
$A > 0$ & $D < A + B$ & $C >$ tangent slope	one or two roots
$A < 0$ & $D < A + B$ & any C	one root
$A < 0$ & $D > A + B$ & $C <$ tangent slope	one or two roots

Table 4.2.1 Root Existence for Functions of Type III

conditions
$A > 0$ & $C > 0$ & $(A + B) \le (C + D)$ & $A > C$
$A > 0$ & $C > 0$ & $(A + B) \ge (C + D)$ & $A < C$
$A > 0$ & $C < 0$ & $(A + B) \le (C + D)$
$A < 0$ & $C > 0$ & $(A + B) \ge (C + D)$
$A < 0$ & $C < 0$ & $(A + B) \le (C + D)$ & $A > C$
$A < 0$ & $C < 0$ & $(A + B) \ge (C + D)$ & $A < C$

Table 4.2.2 Root Existence for Functions of Type IV

4.3. Iteration Definability

Table 4.3.1 summarizes the basic definability results of this chapter. In this section we shall determine for which cases L is definable, and give an expression for the least nonnegative root of h. In the nondefinable cases, we present some special assumptions which yield definability. Not all of these conditions can be established at compile time, some need to be checked at run time.

4.3.1. Case When R0 is not [illegible]

Definability is determined by our ability of finding expressions for the least nonnegative root of h. From Section 4.2 we know that only when h is of type I or of type II we have expressions for the roots. This sets the limits of

	$a < -1$	$a = -1$	$-1 < a < 0$	$a = 0$	$0 < a < 1$	$a = 1$	$a > 1$
$c < -1$	ND	D^{log}	ND	D^{log}	ND	ND	ND
$c = -1$	D^{log}	D	D^{log}	D	D^{log}	D	D^{log}
$-1 < c < 0$	ND	D^{log}	ND	D	ND	ND	ND
$c = 0$	D^{log}	D	D	D	D^{log}	D	D^{log}
$0 < c < 1$	ND	D^{log}	ND	D^{log}	ND	ND	ND
$c = 1$	ND	D	ND	D	ND	D	ND
$c > 1$	ND	D^{log}	ND	D^{log}	ND	ND	ND

Table 4.3.1 Definability Properties of L.

ND means non definable, D^{log} means definable using logarithms,
D means definable without using logarithms.

definability for L.

Theorem 4.3.1

The definable cases are those corresponding to a I or a II entry in Table 4.3.2. The entry indicates the type of h for each case.

Proof

By observing the functions which describe $x[k]$ and $y[k]$.

■

	$a<-1$	$a=-1$	$-1<a<0$	$a=0$	$0<a<1$	$a=1$	$a>1$
$c<-1$		II		II			
$c=-1$	II	I	II	I	II	I	II
$-1<c<0$		II		II			
$c=0$	II	I	II	I	II	I	II
$0<c<1$		II		II			
$c=1$		I		I		I	
$c>1$		II		II			

Table 4.3.2 Types of Definability

Theorem 4.3.2

The expressions arising from type I cases are those represented in Table 4.3.3.

Proof

All of the expressions shown in Table 4.3.3 are established by analyzing the effect of the actions of L on the values of the variables x and y. As an illustrative case, we shall show how to derive the expression for $c = 1$ and $a = -1$. The other cases neither require more work nor different methods.

When $c = 1$ and $a = -1$ we have $x[0] = x[\alpha]$, $x[1] = -x[\alpha] + b$, $x[2] = x[\alpha]$, and in general, for any integer k, $x[2k] = x[\alpha]$, $x[2k+1] = -x[\alpha] + b$. For y, however, the values are: $y[0] = y[\alpha]$, $y[1] = y[\alpha] + d$, $y[2] = y[\alpha] + 2d$, and in general, for any integer k, $y[k] = y[\alpha] + kd$. Clearly, a change of sign in b will occur for either an even or an odd k. The cases are respectively described by

$$h(2k) = x[2k] - y[2k] = 0 = x[\alpha] - y[\alpha] - 2kd \qquad \text{and}$$

$$h(2k+1) = x[2k+1] - y[2k+1] = 0 = -x[\alpha] + b - (2k+1)d = -x[\alpha] + b - d - 2kd .$$

The smallest root of h(k) is given by

	$a = -1$	$a = 0$	$a = 1$
$c=-1$	$\#x\,RO\,y = 1$	$\#x\,RO\,y = \begin{cases} 1 & \text{if } b\,RO(-y[\alpha]+d) \text{ is false} \\ 2 & \text{otherwise} \end{cases}$	$\min\left(\left\lceil \frac{x[\alpha]-y[\alpha]}{-2b} \right\rceil, \left\lceil \frac{x[\alpha]+y[\alpha]+b-d}{-2b} \right\rceil\right)$
$c=0$	$\#x\,RO\,y = \begin{cases} 1 & \text{if } (-x[\alpha]+b)RO\,d \text{ is false} \\ 2 & \text{otherwise} \end{cases}$	$\#x\,RO\,y = 1$	$\left\lceil \frac{x[\alpha]-d}{-b} \right\rceil$
$c=1$	$\min\left(\left\lceil \frac{x[\alpha]-y[\alpha]}{2d} \right\rceil, \left\lceil \frac{-x[\alpha]-y[\alpha]+b-d}{2d} \right\rceil\right)$	$\left\lceil \frac{b-y[\alpha]}{d} \right\rceil$	$\left\lceil \frac{x[\alpha]-y[\alpha]}{b-d} \right\rceil$
$c = 0$ & $-1 < a < 0$	$\#x\,RO\,y = \begin{cases} 1 & \text{if } (ax[\alpha]+b)RO\,d \text{ is false} \\ 2 & \text{otherwise} \end{cases}$		
$a = 0$ & $-1 < c < 0$	$\#x\,RO\,y = \begin{cases} 1 & \text{if } b\,RO(cx[\alpha]+d) \text{ is false} \\ 2 & \text{otherwise} \end{cases}$		

Table 4.3.3 Expressions From Type I Cases

$$\min\left\{ \frac{x[\alpha]-y[\alpha]}{2d}, \frac{-x[\alpha]-y[\alpha]+b-d}{2d} \right\} .$$

■

When $|a| > 1$, let f_i be that function among f_p and f_m which is increasing, and f_d that which decreases. Similarly we define g_i and g_d. These symbols are used in Table 4.3.4.

Theorem 4.3.3

The expressions arising from type II cases are those represented in Table 4.3.4.

Proof

$c = 0$ & $0 < a < 1$ or $c = 0$ & $a > 1$	$\#xROy = \begin{cases} 1 & \text{if } x[\alpha] = \dfrac{b}{1-a} \\ \left\lceil \dfrac{\log\left(\dfrac{d(a-1)+b}{x[\alpha](a-1)+b}\right)}{\log(a)} \right\rceil & \text{otherwise} \end{cases}$
$a = 0$ & $0 < c < 1$ or $a = 0$ & $c > 1$	$\#xROy = \begin{cases} 1 & \text{if } y[\alpha] = \dfrac{d}{1-c} \\ \left\lceil \dfrac{\log\left(\dfrac{b(c-1)+d}{y[\alpha](c-1)+d}\right)}{\log(c)} \right\rceil & \text{otherwise} \end{cases}$
$c = 0$ & $a < -1$	$f_i(r) = d$ if $d \geq \dfrac{b}{1-a}$ $\quad f_d(r) = d$ if $d < \dfrac{b}{1-a}$
$a = 0$ & $c < -1$	$g_i(r) = b$ if $b \geq \dfrac{d}{1-c}$ $\quad g_d(r) = b$ if $b < \dfrac{d}{1-c}$
$c = -1$ & $a < -1$ or $c = -1$ & $-1 < a < 0$	$\min_{r}\{f_m(r) = -y[\alpha] + d\, ,\ f_p(r) = y[\alpha]\}$
$a = -1$ & $c > 1$ or $c = -1$ & $0 < a < 1$	$\min_{r}\{f_p(r) = -y[\alpha] + d\, ,\ f_p(r) = y[\alpha]\}$
$a = -1$ & $c < -1$ or $a = -1$ & $-1 < c < 0$	$\min_{r}\{g_m(r) = -x[\alpha] + d\, ,\ g_p(r) = x[\alpha]\}$
$a = -1$ & $c > 1$ or $a = -1$ & $0 < c < 1$	$\min_{r}\{g_p(r) = -x[\alpha] + d\, ,\ g_p(r) = x[\alpha]\}$

Table 4.3.4 Expressions From Type II Cases

All of the above expressions are established by analyzing the effect of the actions of L on the values of the variables x and y. As an illustrative case we shall show how to derive the expressions for the case $c = -1$ and $a < -1$.

In this case we have $x[0] = x[\alpha] = f_p(0)$, $x[1] = f_m(1)$, $x[2] = f_p(2)$, $x[3] = f_m(3)$, and in general, for all integers k, $x[2k] = f_p(2k)$, $x[2k+1] = f_m(2k+1)$. For y, however, the values are given by $y[0] = y[\alpha]$, $y[1] = -y[\alpha] + d$, $y[2] = y[\alpha]$, $y[3] = -y[\alpha] + d$, and in general, for all integers k, $y[2k] = y[\alpha]$, $y[2k+1] = -y[\alpha] + d$. As both $|f_p|$ and $|f_m|$ are unbounded monotonically increasing functions, there will be at least one solution r for $f_m(r) = -y[\alpha] + d$ or $f_p(r) = y[\alpha]$.

If f_m has a root r, we find the least odd integer $(2k + 1) \geq r$. If f_p has a root r, we find the least even integer $2k \geq r$. We then find $\#xROy$ according to Theorem 3.2.1, using the minimum of the above two numbers. If only one root exists, we use the corresponding minimum integer found and Theorem 3.2.1.

■

4.3.1.1. More Hypotheses Which Yield Definability

In all cases when h is of type III or of type IV, there are some instances in which it is possible to obtain an expression for the roots. They arise either from special values that the control variables x, y have when entering the iteration L, or from relationships between a and c. The result of these simplifications is that functions of type I or II are obtained.

Functions of Type III

In this case $h(r) = a^r A + B - (rd + y[\alpha])$ where $A = x[\alpha] + \dfrac{b}{a-1}$ and $B = \dfrac{b}{1-a}$. There are only two simplifications which may occur:

(1) $x[a] = \frac{b}{a-1}$

Then, $h(r) = -rd + B - y[a]$, which is of type I.

(2) $d = 0$

Then, $h(r) = a^r A + B - y[a]$, which is of type II.

Functions of Type IV

In this case, $h(r) = a^r A + B - (c^r C + D)$, where $A = x[a] + \frac{b}{a-1}$, $B = \frac{b}{1-a}$, $C = y[a] + \frac{d}{c-1}$, and $D = \frac{d}{1-c}$. The only four simplifications which may occur are the following:

(1) $x[a] = \frac{b}{a-1}$

Then, $h(r) = -c^r C + B - D$, which is of type II.

(2) $y[a] = \frac{d}{c-1}$

Then, $h(r) = a^r A + B - D$, which is of type II.

(3) $a = c$

Then, $h(r) = a^r(A - C) + B - D$, which is of type II.

(3) $B = D$

Then, $h(r) = a^r A - c^r C$, and the root r for $h(r) = 0$ is given by $r = \frac{\log\left(\frac{C}{A}\right)}{\log\left(\frac{a}{c}\right)}$.

4.3.2. Case When R0 is ≠

As was pointed out in Section 3.2, when analyzing the case of ≠ we are required to find an integer root of h. When halting is being assumed, this amounts to searching through the distinct roots of h, at least one of which

will be integer. In Section 4.2 we saw that in all definable cases for iterations satisfying the hypotheses of this chapter, b has at most two roots. This certainly limits the amount of searching one needs to do. Moreover, in Section 4.5 we shall give conditions which delimit the the range of the search to be performed.

In many of the nondefinable cases the function h has infinitely many roots. In fact, this condition may also be dependent on the values of $x[a]$ and $y[a]$. In Section 4.4 we see several cases where this occurs. What interests us now is the fact that, if h changes sign infinitely many times, unless one has an a priori method to determine if h will have an integer root, a search for it may be endless. This is the main difficulty when dealing with the relational operator ≠. As no expressions for the roots of systems of two exponential equations exist in general, only a case by case analysis can provide an answer about the existence of integer roots. Determining $\#\varphi$ in this way is not acceptable. Given our hypothesis that programs halt, we would use the skeleton approach to deal with this case.

4.4. Stability of Predicates

Changes of truth values of $x R0 y$ are associated with changes in the sign of h. Thus the study of h's behavior will also determine the sequences of truth values which may occur in iterations satisfying the hypotheses of this chapter. Theorem 3.2.2 tells us that the monotonicity of h is the condition which yields stability. Given the monotonicity of linear functions, it is not surprising then that many cases will have stable predicates. However, non-stable cases do exist, but they have remarkably simple patterns of truth values. This information can be used advantageously when evaluating $\#\varphi$ for non atomic predicates φ. There are several cases where the behavior of the predicates can be determined at compile time.

Table 4.4.1 depicts abbreviated versions of the patterns of truth values which may occur when the predicate $x$RO$y$ is not $x \neq y$. The justification of each entry is, once again, based on an analysis of the behavior of the associated function h. The behavior of h is determined by the relative growths of the functions f and g.

Let us now analyze one entry of the table in complete detail.

Case when a = -1 and -1 < c < 0.

In this case we have $x[2k] = x[\alpha]$, and $x[2k+1] = -x[\alpha] + b$. The values for $y[k]$, on the other hand, are given by: $y[2k] = g_p(2k)$ and $y[2k+1] = g_m(2k+1)$. We know that both g_p and g_m are monotonic and have limits, when $k \to \infty$, equal to $\frac{d}{1-c}$. We then have that

$$x[k]\text{RO}y[k] = \begin{cases} x[\alpha]\text{RO}g_p(k) & \text{when k is even} \\ (-x[\alpha] + b)\text{RO}g_m(k) & \text{when k is odd.} \end{cases}$$

As g_p and g_m are monotonic, the sequences of truth values generated by them will be stable, i.e., there will be at most one change in the truth value. As we assume that L's T-branch will be traversed at least once, we have that $x[\alpha]$RO$y[\alpha]$ is true. Thus, the possible sequences of truth values generated by $x[\alpha]$RO$g_p(k)$ are:

$$T \cdots T \begin{cases} TTT \cdots & \text{if } x[\alpha]\text{RO}\frac{d}{1-c} \text{ is true} \\ FFF \cdots & \text{if } x[\alpha]\text{RO}\frac{d}{1-c} \text{ is false} \end{cases}.$$

The possible sequences of truth values generated by $(-x[\alpha] + b)$RO$g_m(k)$ can be obtained as follows. Let $P = (-x[\alpha] + b)\text{RO}g_m(1)$ and $Q = (-x[\alpha] + b)\text{RO}\frac{d}{1-c}$; then,

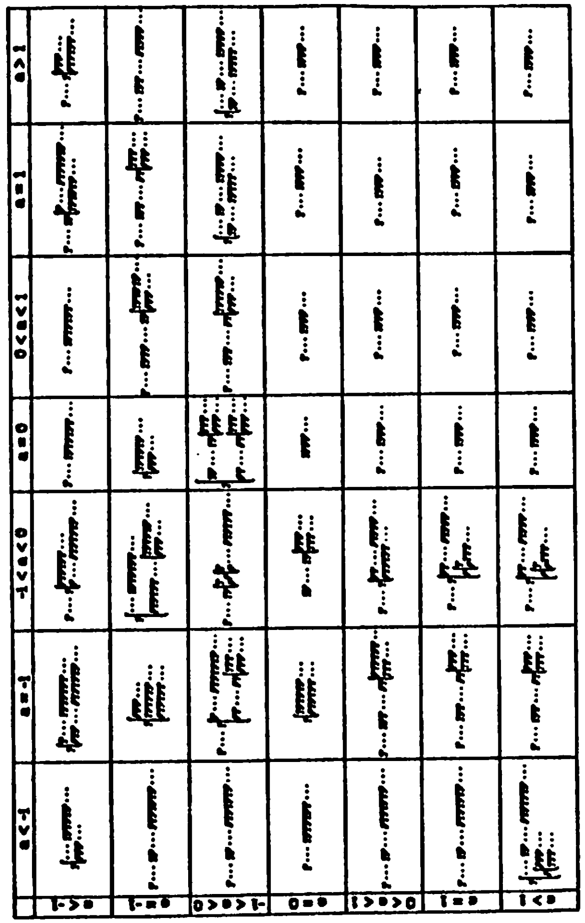

Table 4.4.1 Patterns of Truth Values For predicates RO not ≠.

$$\begin{cases} T\cdots T\begin{cases} TTT\cdots & \text{if P is true and Q is true} \\ FFF\cdots & \text{if P is true and Q is false} \end{cases} \\ F\cdots F\begin{cases} TTT\cdots & \text{if P is false and Q is true} \\ FFF\cdots & \text{if P is false and Q is false.} \end{cases} \end{cases}$$

All the sequences of truth values which b may produce are obtained by an appropriate selection of alternative truth values from these two sequences. As we are assuming halting programs, there is only one case excluded from the eight combinations, namely, that which produces the sequence $TTT\cdots$. Thus, the halting condition for this case, i.e., when a = -1 and -1 < c < 0, is that

$$(x[\alpha]\mathrm{RO}\frac{d}{1-c}) \;\&\; ((-x[\alpha]+b)\mathrm{RO}(cy[\alpha]+d)) \;\&\; ((-x[\alpha]+b)\mathrm{RO}(\frac{d}{1-c}))$$

is false, where we have made use of the equality $g_m(1) = cy[\alpha] + d$. This condition will be used in Section 4.5.

Before analyzing the remaining seven cases, we shall indicate how one determines the lengths of the initial sequences of "trues" or "falses" obtained from g_p and g_m. The whole idea is that we are dealing with functions of type II, and so we may find an expression for the root. The floor of this root gives us the length of the initial sequence of truth values.

In particular, let us analyze the case of g_p. We assume that $x[\alpha]\mathrm{RO}\frac{d}{1-c}$ is false and we want to determine the number of consecutive times $x[\alpha]\mathrm{RO}g_p(k)$ will evaluate to true. As we assume that $x[\alpha]\mathrm{RO}g_p(0)$ is true, recalling that $g_p(0) = y[\alpha]$, our problem is equivalent to finding the least k for which the function $g_p(k) - x[k]$ changes sign. The root for this function is given by

$$r = \frac{\log\left(\dfrac{x[\alpha] + \dfrac{d}{1-c}}{y[\alpha] + \dfrac{d}{1-c}}\right)}{\log(|c|)}.$$

Thus, the number of successive times that $x[\alpha]\mathrm{RO}g_p(k)$ will evaluate to true is $\lfloor r\rfloor$. An analogous analysis can be carried out for g_m.

We now analyze the remaining seven alternatives: let $P = (x[\alpha]\mathrm{RO}\frac{d}{1-c})$, $Q = ((-x[\alpha] + b)\mathrm{RO}(\frac{d}{1-c}))$ and $R = ((-x[\alpha] + b)\mathrm{RO}(cy[\alpha] + d))$.

(i) P is true, Q is true, R is false

we obtain: $TFTF\cdots TFTTT\cdots$

(ii) P is true, Q is false, R is false

we obtain: $TFTFTF\cdots$

(iii) P is true, Q is false, R is true

we obtain: $T\cdots TTFTFTF\cdots$

(iv) P is false, Q is true, R is false

we obtain: $TFTF\cdots TF\begin{cases} F\begin{cases} F\cdots FFTFTFT\cdots \\ T\cdots TFTFTFT\cdots \end{cases} \\ T\cdots TFTFTFT\cdots \end{cases}$

(v) P is false, Q is false, R is false

we obtain: $TFTF\cdots TFFFF\cdots$

(vi) P is false, Q is true, R is true

we obtain: $T\cdots TFTFTFT\cdots$

(vii) P is false, Q is false, R is true

we obtain:

$$T \cdots T\begin{cases} TF \cdots TFFFF \cdots \\ F\begin{cases} TF \cdots TFFFF \cdots \\ FFF \cdots \end{cases} \end{cases}$$

The above seven cases can be summarized as follows:

$$T \cdots T\begin{cases} FTFT \cdots FT\begin{cases} TTT \cdots \\ FFF \cdots \end{cases} \\ FTFTFT \cdots \end{cases}$$

With these techniques, one can determine, at run time, the exact pattern of truth values that h will have, even in the nonstable cases. The reduction to subcases involving functions of type II is what allows us to do so.

We should remark that when the relational operator is not $\neq$, all stable cases are those where h has exactly one root. It needs to have one because of halting, and it can't have more than one because of the change of sign. On the other hand, when the relational operator is $\neq$, the condition on the roots of h is that there exist exactly one integer root. Noninteger roots may exist in abundance.

4.5. Halting Conditions

In Section 4.2 we listed the cases where h would have roots. The cases were given in terms of conditions for the coefficients A, B, C and D. These conditions naturally translate themselves into conditions involving $x[\alpha]$, $y[\alpha]$, a, b, c and d. From them, as we have just done in Section 4.4, we may deduce conditions on $x[\alpha]$ and $y[\alpha]$ which would insure halting, given that a, b, c and d are assumed not to change their value in L.

Moreover, it should also be clear that, when the values of $x[\alpha]$ and $y[\alpha]$ are available, some nondefinable cases become definable, as seen in Section 4.3.1.1, and so, even in those cases, the expression of $\#x\,\mathrm{RO}\,y$ can be obtained at run time.

When our relational operator is not $\neq$, there are several cases when halting can be established at compile time. For instance, assume $a < -1$. We know that the values of $x[k]$ (when $x[\alpha] \neq \frac{b}{1-a}$) will be unbounded. Moreover, consecutive values will have, for k sufficiently large, opposite signs. Then, if $c \geq -1$, all of these iterations will halt because either the values of $y[k]$ are bounded or they all have, for k sufficiently large, the same sign. This argument is also valid, with the roles of x and y interchanged, for the case when $c < -1$ and $a > -1$.

The two most difficult cases for halting conditions are encountered when dealing with functions h of type III and of type IV. We shall now give criteria which reduces the scope of a search for the roots.

Let us analyze the case $a > 1$ and $c = 1$. Our function h is given by $h(r) = a^r A + B - (rd + D)$, where $A = x[\alpha] + \frac{b}{a-1}$, $B = \frac{b}{1-a}$ and $D = y[\alpha]$. We have seen that this case is non definable. However, we may bound the smallest nonnegative root using the following observation: if at the origen the straight line is above the exponential, then 0 is a lower bound. Otherwise, the straight line $y = rd + D$ must intersect the exponential function $y = a^r A + B$ for the first time at a point where the exponential's growth rate is smaller than the slope of the line. This observation is true because, if they have not met by that time, the growth of the exponential will be larger than that of the straight line and thus the values of the latter will never be able to reach those of the exponential.

An upper bound on the least nonnegative root of h is given by a formalization of the above remark. The derivative of $h(r)$ is $h'(r) = \ln(a)Aa^r$. The slope of the straight line is d. So we find r such that $h'(r) = d$, and this is our upper bound for the least nonnegative root of h. This r is given by

$$r = \frac{\log\left(\frac{d}{\ln(a)A}\right)}{\log(a)} .$$

This r may have to be determined at run time, because A depends on $x[\alpha]$. If $x[\alpha]$ were known at compile time, then r could be determined at compile time. Moreover, if r is small, instead of finding all the corresponding roots for our functions, one may decide to use the skeleton approach for the iteration.

Consider now the case $a > 1$ and $c > 1$. We have

$h(r) = a^r A + B - (c^r C + D)$, where $A = x[\alpha] + \frac{b}{a-1}$, $B = \frac{b}{1-a}$, $C = y[\alpha] + \frac{d}{c-1}$, and $D = \frac{d}{1-c}$. When we are not in one of the four conditions listed in Section 4.3.1.1, we have to bound the root. We look for upper bounds as tight as possible. When we have the upper bound, we perform a "midpoint search" to find the roots. This search is much faster than a sequential search.

Equation $a^r A - c^r C = D - B$ is equivalent to $\frac{A}{C}\left(\frac{a}{c}\right)^r = 1 + \frac{D-B}{Cc^r}$. As $c > 1$, we have that $\lim_{r\to\infty} \frac{D-B}{Cc^r} = 0$. There are two cases to consider:

(i) $\frac{D-B}{C} < 0$

In this case the smallest value which may be achieved by the left hand side of our equivalence is $1 + \frac{D-B}{C}$. Thus, we will be passed the root when

$$\frac{A}{C}\left(\frac{a}{c}\right)^r = 1 + \frac{D-B}{C},$$

i.e., when

$$r = \frac{\log\left(\frac{C}{A}\left(1 + \frac{D-B}{C}\right)\right)}{\log\left(\frac{a}{c}\right)} .$$

(ii) $\frac{D-B}{C} > 0$

In this case the smallest value which may be achieved by the left hand side of our equivalence is 1. Thus we will be passed the root when $\frac{A}{C}\left(\frac{a}{c}\right)^r = 1$, i.e., when $r = \frac{\log\left(\frac{C}{A}\right)}{\log\left(\frac{a}{c}\right)}$.

Tighter bounds may be found by using better values for the right hand side of our equivalence.

There is yet another consideration to be made about non definable cases. The range of values that their underlying exponential functions may take is quite restricted. As most programming languages only support a finite number of distinct values for variables of type numeric, once they are exceeded, overflow (or underflow) conditions will halt the computation. As we assume *bona fide* halting programs, the values of the variables should always stay within the predetermined ranges.

Let us consider $g(r) = Aa^r + B$ when r is of type integer. Assume further that the underlying hardware has 32 bit words. Then, the largest unsigned integer we may represent is $2^{32} - 1$.

The function g will grow at its slowest rate when A is smallest, and B is the largest possible negative number. Assuming these conditions, g is transformed to $g(r) = 2^{32}a^r - 3^{32}$ and we may easily find the value of r, as a function of a, for which $g(r) = 2^{32}$. Namely, $r = \frac{65\log(2)}{\log(a)}$. So, when $a \geq 2$, we have that $r \leq 65$. That is, an iteration under these hypotheses does not traverse consecutively its T-branch more than 65 times.

This hardware dependent information, together with compile time information about the nestedness of iterations, should be used when determining what strategy to follow in the analysis of a given program.

4.6. Summary

In this chapter we have seen how to determine all the information needed for our analysis in the case of an iteration in which the actions on the control variables are linear functions. For example, we determined all the definable cases. We have given expressions for h which allow us to find $\#\varphi$ in all the nondefinable cases. We have also given bounds for the interval of numbers in which the search for the roots of h must be made. Moreover, halting conditions for all cases were also given. We were even able to find the sequences of truth values that predicates have when the actions of the iteration on the control variables are linear functions.

CHAPTER 5

Finding Closed Forms

In Section 2.5, the subject of closed forms for variables in iterations was first introduced. It was seen that closed forms must exist for all control variables in a program if we are to find our best possible performance representation of the program. Moreover, for definable programs, the special terms describing the closed forms cannot contain any occurrence of n-place function symbols f. We shall call this latter type of closed forms *definable closed forms*. In Chapter 4, we have analyzed a case where finding closed forms for control variables is no problem. Indeed, they always were definable closed forms.

The purpose of this chapter is five-fold:

(1) to reduce the problem of finding closed forms for irreducible iterations to that of solving recurrence relations;

(2) to describe, discuss and analyze three known decision procedures for finding definable closed forms for some families of recurrence relations;

(3) to present table-driven methods for finding definable closed forms;

(4) to discuss the problem of closed forms in the context of iterations with multiple inner paths; and

(5) to discuss the problem of closed forms in the context of nested iterations.

5.1. Irreducible Iterations and Recurrence Relations

An irreducible iteration is an iteration $L = <B, \varphi, \alpha, \beta>$ whose body B is a basic block of statements. As our programs are assumed not to be self-modifying, the action of L on a variable x may be described by a fixed function g. The specific form of g will depend on the kind of action L performs on x.

We let $x[0]$ denote the value that the variable x has when entering the iteration L for the first time, i.e., $x[0] = x[\alpha]$. Then, $x[k+1]$, i.e., the value x has after the T-branch of L has been traversed k+1 consecutive times, can be written as $x[k+1] = g(x[k], k)$. g is determined in a unique way by the statements in B.

The triple $<x[0], x[k], x[k+1]>$, constitutes a first order recurrence relation with boundary condition $x[0]$.

In Chapter 4, we studied the case where, for all control variables x, the relationship between $x[k+1]$ and $x[k]$ does not depend on the iteration index k and, moreover, the relationship between $x[k+1]$ and $x[k]$ is linear. In this case, definable closed forms are easy to obtain.

Whenever we have that the relationship between $x[k+1]$ and $x[k]$ does not depend on the iteration index k, we may express $x[k+1]$ in the following way:

$$x[k+1] = g(x[k]) = g^k(x[0]) .$$

In this case, finding definable closed forms is the same as finding a term $\tau(x,k)$ which describes $g^k(x)$.

In the work of Cheatham [Che78], we see that there are situations when one wants to find closed forms for parametric recurrence relations. In their most general form, parametric recurrence relations can be expressed by,

$$A_{k+1}(x) = g(A_k(h(x,k)), x, k) , \text{ where } k > 0 .$$

In the same paper it is proven that solving recurrence relations of this kind may be reduced to solving at most two parametric recurrence relations of the form

$$A_{k+1}(x) = g(A_k(x), x, k) , \text{ where } k > 0 .$$

Relations of the latter type are easier to solve because the "parameter" x does not change throughout the iterations. A pattern-matching table-lookup method for solving this type of recurrence was proposed by Cheatham.

First order recurrence relations have been studied since the last century. The first treatise on the subject was written by G. Boole in 1872 [Boo57]. The mathematical methods proposed for solving them have paralleled those for differential equations. In fact, it has been shown that systems of linear recurrence relations with constant coefficients can always be reduced to a single recurrence relation with constant coefficients in one variable [Lev81]. The reduction can be accomplished in an automatic way using Cramer's rule for solving systems of linear equations.

Once one is dealing with a single recurrence relation with constant coefficients, there are basically two methods for solving it:

(1) Determine the solution of an associated homogeneous recurrence relation by a well defined algorithm. Then determine a particular solution of the full recurrence relation by guessing its form. As in the case of differential equations, the sum of these two solutions is the general solution.

(2) Find a generating function whose coefficients represent the sequence defined by the recurrence relation. The expression for the kth coefficient is the desired solution.

Unfortunately, there is no known way of finding, in an automatic way, expressions for the k^th term of a generating function. Thus, both of the above methods require some kind of human intervention or table look-up.

Recurrence relations with variable coefficients present a whole new spectrum of complications. As in the case of differential equations with variable coefficients, there are no general methods for finding solutions. We shall now analyze a case which covers more than 90% of the modifications made to control variables in the studies of programs we have seen. We shall then see how the decision procedures presented in the next section help us find closed forms for this case.

We assume that the value of the variable x is modified by the two functions f and g as follows: $x[k+1] = g(k)x[k] + f(k)$. Chapter 4 dealt with the special case when both functions, f and g, were constant.

Lemma 5.1.1

If, for all integers $k \geq 0$, $x[k+1] = g(k)x[k] + f(k)$, then

$$x[k+1] = \prod_{t=0}^{k} g(t)x[0] + \prod_{t=1}^{k} g(t)f(0) + \cdots + \prod_{t=k-1}^{k} g(t)f(k-2) + \prod_{t=k}^{k} g(t)f(k-1) + f(k)$$

Proof

By induction on k. For $k = 0$ we have

$$x[1] = g(0)x[0] + f(0) = \prod_{t=0}^{0} g(t)x[0] + f(0)$$

and so our expression holds. Assuming that our expression is true for j, we have

$$x[j+1] = g(j)x[j] + f(j)$$

$$= g(j)\left(\prod_{t=0}^{j-1} g(t)x[0] + \prod_{t=1}^{j-1} g(t)f(0) + \cdots + \prod_{t=j-2}^{j-1} g(t)f(j-3) + \prod_{t=j-1}^{j-1} g(t)f(j-2) + f(j-1)\right)$$

$$+ f(j)$$

$$= \prod_{t=0}^{j} g(t)x[0] + \prod_{t=1}^{j} g(t)f(0) + \cdots + \prod_{t=j-1}^{j} g(t)f(j-2) + \prod_{t=j}^{j} g(t)f(j-1) + f(j) .$$

■

Theorem 5.1.2

If, for all nonnegative integers n, $g(n) \geq 1$, then the existence of a closed form for $\sum_{t=0}^{k} f(t)$ yields an alternative expression for $x[k]$ whose evaluation cost is linear in k.

Proof

$g(j) \geq 1$ implies that

$$\prod_{t=j}^{k} g(t) \geq \prod_{t=j+1}^{k} g(t) .$$

By rearranging terms in the expression obtained for $x[k]$ in Lemma 5.1.1 and using this fact about g, one obtains:

$$x[k+1] = \prod_{t=0}^{k} g(t)x[0] + \left(\prod_{t=1}^{k} g(t) - \prod_{t=2}^{k} g(t)\right)\sum_{t=0}^{0} f(t) + \cdots$$

$$+ \left(\prod_{t=j}^{k} g(t) - \prod_{t=j+1}^{k} g(t)\right)\sum_{t=0}^{j-1} f(t) + \cdots + \left(\prod_{t=k}^{k} g(t) - 1\right)\sum_{t=0}^{k-1} f(t) + \sum_{t=0}^{k} f(t) .$$

This equality can be verified easily by induction. The closed form hypothesis for the sum allows us to obtain each sum with one evaluation. To efficiently obtain the products, one evaluates them from the "last" down to the "first", i.e., shorter products are computed before longer ones. We notice that, in our expression for $x[k+1]$, we need to have only two adjacent products of this kind at the time. Thus, the total evaluation cost is linear in k.

■

The expressions obtained for $x[k+1]$ in Lemma 5.1.1 and in Theorem 5.1.2 provide computational alternatives. The more closed forms one has, the faster the evaluation can get. In fact, if we were also to have a closed form for $\prod_{i=j}^{k} g(i)$, the evaluation time might be even shorter, depending on the complexity of the closed form.

Corollary 5.1.3

If, for all integers n, $g(n) = m$, where $m \geq 1$, i.e., g is the constant function m and m is at least one, then the expression for $z[k+1]$ simplifies to:

$$z[k+1] = m^{k+1}z[0] + \left(m^{k} - m^{k-1}\right)f(0) + \cdots +$$

$$\left(m^{k-j+1} - m^{k-j}\right)\sum_{i=0}^{j} f(i) + \cdots + (m-1)\sum_{i=0}^{k-1} f(i) + \sum_{i=0}^{k} f(i)$$

Proof

By using the expression derived in Theorem 5.1.2 .

■

Thus, definable closed formulae for $\prod_{i=j}^{k} g(i)$ and $\sum_{i=0}^{k} f(i)$ allow us to evaluate $z[k+1]$ with cost linear in k. Depending on the complexity of the closed forms, evaluating $z[k+1]$ may be done faster using the skeleton. Finding a closed form for $z[k]$, however, is much harder. We remark that we have made no assumptions on f.

When dealing with product forms, the usage of logarithms may help us obtain closed forms. If $\pi = \prod_{i=0}^{k} g(i)$, then $\log(\pi) = \sum_{i=0}^{k} \log(g(i))$. If there is a closed form for the latter, then the value of the former may be obtained in a pointwise manner by using antilogarithms.

5.2. Three Decision Procedures for Finding Closed Forms

The three procedures we shall analyze present different approaches to the problem. One of the differences has to do with how the closed form is expressed. Moenck's procedure makes use of a set of functions, the

polygamma functions, to express closed forms [Moe77]. Karr's procedure requires the specification of which formal symbol(s) should be used to express the closed form [Kar79], and then it decides if the sum is a rational function of these symbols or not.

On the other hand, Gosper's procedure does not do anything of this sort, but the answers it provides may contain expressions which require iterative methods to evaluate [Gos78].

5.2.1. Moenck's Procedure

Moenck considers the problem of summing polynomials and rational functions. His approach is to follow, as closely as possible, the techniques and approach used for integration of polynomials of rational functions. He notices that the correct analogy to difference algebra of the differentiation operator D in differential algebra (i.e.: $Dx^n = nx^{n-1}$) is the difference operator Δ acting on the the factorial function $[x]_n$. Factorial functions are defined as $[x]_n = x(x-1)(x-2)\cdots(x-n+1)$. The difference of a factorial is

$$\Delta[x]_n = n[x]_{n-1} \ .$$

Moenck's approach for finding sums of polynomials is to represent them in terms of factorial functions and then use this new representation to find their sum. If

$$f(x) = \sum_{i=0}^{n} \frac{[x]_i}{i!}\Delta^i f(0)$$

then

$$\Delta^{-1}f(x) = \sum_{i=0}^{n} \frac{[x]_{i+1}}{(i+1)!}\Delta^i f(0) = \sum_{i=0}^{n} \binom{x}{i+1} \Delta^i f(0) \,.$$

To actually compute the expression for a given polynomial, one first needs to compute the difference table for all appropriate entries (as determined by the value of n in $\Delta^{-1}f(x)$) and then, if one wants a power representation instead of a factorial representation, one needs to use the explicit definitions of the factorial functions in terms of polynomials. As far as summing polynomials, this method requires more work than the straightforward one of distributing the sums across the powers of the variables and using the well known recurrence relations which express $\sum_{i=1}^{n} x^i$ in terms of the sums of lower degree. However, Moenck's techniques provide a way to generalize this procedure to rational functions.

The factorial operator on a function f, $[f(x)]_k$, is defined by

$$[f(x)]_k = f(x)f(x-1)\cdots f(x-k+1) \text{ for } \quad k>0 .$$

Then, it is extended to all integers k by defining $[f(x)]_0 = 1$ and asserting that

$$[f(x)]_k = [f(x)]_j[f(x-j)]_{k-j}$$

is an identity.

To carry on the analogy with partial fraction decomposition and integration by parts, Moenck needs to find "shift free" decompositions of rational functions. "shift free" means that the members in a product of functions are not shifts, to any power, of another member of the product.

Thus, Moenck's procedure begins with the following algorithm which produces shift-free rational expressions: Given a rational function $\frac{A(x)}{S(x)}$

1) Form S(x+k), where k is a new variable.

2) Compute the resultant with respect to k: Res(S(x+k), S(x)) = R(k) .

3) Test for integer roots of R(k): these will disclose any k's with non trivial GCD's of the form GCD(S(x), $E^kS(x)$).

4) Apply Stirling's method to convert the rational function $\frac{A(x)}{S(x)}$ into a factorial denominator: this is done by multiplying the numerator by appropriate factors of the factorial function which are missing in the denominator.

5) Proceed to form a complete shift-free partial fraction decomposition, i.e., $\frac{A(x)}{S(x)} = \sum_{i=1}^{k}\sum_{j=1}^{i}\frac{A_{i,j}(x)}{[s_i]_j}$, where each $[s_i]_j$ is shift-free, and is found by the above steps.

Then the algorithm proceeds to use summation by parts, applied to each term of the shift-free partial fraction decomposition. To express the result for the terms of the form $\frac{a_i}{(x-b_i)^{j(i)}}$, i.e., the transcendental part, the polygamma functions $\psi_m(x)$ are introduced. We recall that

$$\psi_m(x) = D^m \log(\Gamma(x+1)), \quad m > 0 .$$

Polygamma functions behave nicely with the difference operator Δ, since

$$\Delta\psi_m(x) = D^{m-1}\left(\frac{1}{x+1}\right) = \frac{(-1)^{m-1}(m-1)!}{(x+1)^m}$$

Therefore, if

$$\frac{A(x)}{B(x)} = \sum_{i=1}^{k}\frac{a_i}{(x-b_i)^{j(i)}} ,$$

where j(i) is the multiplicity of the root b_i, using the polygamma functions the summation can be written as

$$\Delta^{-1}\frac{A(x)}{B(x)} = \sum_{i=1}^{k}\frac{a_i(-1)^{j(i)-1}}{(j(i)-1)!}\psi_{j(i)}\left(x-b_i-1\right) .$$

With this expression, one can find pointwise values by evaluating appropriately the polygamma functions. It should be noticed, however, that the evaluation may be quite expensive if the transcendental part is long.

Moreover, one is not able to determine whether a transcendental part exists until the full shift-free partial function decomposition has been carried out.

This procedure provides us with a method to find closed forms for the action of an iteration L on a variable x of the following kind:

$$x[k+1] = x[k] + f(k),$$

where f is a rational function. In Section 5.1 we have also seen how closed forms for sums can be used in more general actions of iterations on variables.

It should be pointed out that most of Moenck's procedure can be traced back to Jordan's book on Finite Differences [Jor50].

5.2.2. Gosper's Procedure

Given a summand a_i, the "indefinite sum" S(n) is determined (within an additive constant) by $\sum_{i=1}^{m} a_i = S(m) - S(0)$, or, equivalently, by

$$a_i = S(i) - S(i-1) \qquad [1]$$

When S(n) has the additional property that $\frac{S(n)}{S(n-1)}$ is a rational function of n, Gosper's procedure finds S(n).

Gosper's technique consists of performing a particular change of variables which reduces $a_i = S(i) - S(i-1)$ to a system of linear equations. This system is consistent if $\frac{S(n)}{S(n-1)}$ is a rational function of n. When the system is consistent, by solving it one can find the coefficients of a polynomial which yields the closed form for S(n). The essence of the change of variables is that, as in Moenck's algorithm, a shift-free factor is sought to exist in a decomposition of a rational function. We shall now describe the method in some detail.

Assume that our summands a_n are all nonzero. Then, when $\frac{S(n)}{S(n-1)}$ is a rational function of n,

$$\frac{a_n}{a_{n-1}} = \frac{S(n) - S(n-1)}{S(n-1) - S(n-2)} = \frac{\frac{S(n)}{S(n-1)} - 1}{1 - \frac{S(n-2)}{S(n-1)}} \qquad [2]$$

must also be a rational function of n. We want to express the ratio [2] as follows:

$$\frac{a_n}{a_{n-1}} = \frac{p_n}{p_{n-1}} \frac{q_n}{r_{n-1}} \qquad [3]$$

where p_n, q_n, and r_n are polynomials in n subject to the condition that q_n and r_n are shift free relatively prime. Thus, we require that $GCD(q_n, r_{n+j}) = 1$ for all nonnegative integers j. We have seen in Section 5.2.1 that this can be achieved by computing the nonnegative roots of the resultant of q_n and r_n and by performing a change of variables.

In fact, if $GCD(q_n, r_{n+j}) = g(n)$, then this common factor can be eliminated with the transformation

$$q'_n \leftarrow \frac{q_n}{g(n)}, \quad r'_n \leftarrow \frac{r_n}{g(n-j)}$$
$$p'_n \leftarrow p_n g(n) g(n-1) \cdots g(n-j+1) \ ,$$

which leaves ratio [3] unchanged.

One now expresses S(n) by

$$S(n) = \frac{q_{n+1}}{p_n} f(n) a_n, \qquad [4]$$

where f(n) is to be determined. By using equation [1],

$$f(n) = \frac{p_n}{q_{n+1}} \frac{S(n)}{S(n) - S(n-1)} = \frac{p_n}{q_{n+1}} \frac{1}{1 - \frac{S(n-1)}{S(n)}} ;$$

thus, f(n) is a rational function of n whenever $\frac{S(n)}{S(n-1)}$ is. By substituting equation [4] into [1], one gets

$$a_n = \frac{q_{n+1}}{p_n} f(n) a_n - \frac{q_n}{p_{n-1}} f(n-1) a_{n-1} \cdot$$

Multiplying this by $\frac{p_n}{a_n}$ and using equation [3], one has

$$p_n = q_{n+1} f(n) - r_n f(n-1) \quad , \qquad [5]$$

the functional equation for f.

Theorem 5.2.2.1 (Gosper)

If $\frac{S(n)}{S(n-1)}$ is a rational function of n, then f(n) is a polynomial.

Proof

See [Gos78].

∎

Then, because of equation [4], all that remains to be done is to look for a polynomial f(n) satisfying equation [5], given p_n, q_n, and r_n. This is done by considering two cases according to the relationship between the degrees of the polynomials $(q_{n+1} + r_n)$ and $(q_{n+1} - r_n)$.

The closed forms which this procedure finds may contain expressions that require iterative methods for their evaluation. Nevertheless, the expressions make no use of functions like the polygamma ones. The class of indefinite sums for which this procedure produces closed forms contains many families which can not be solved using Moenck's procedure. For example, Gosper's procedure can sum

$$\sum_{n=1}^{n} \frac{\prod_{j=1}^{n-1} \left(aj^3 + bj^2 + cj + d\right)}{\prod_{j=1}^{n} \left(aj^3 + bj^2 + cj + e\right)},$$

while Moenck's procedure cannot. However, given that Moenck's procedure expresses part of its answers using polygamma functions, it produces

expressions for some sums which Gosper's procedure considers unsummable. One example of this is the case of $\sum_{i=1}^{n} \frac{1}{i}$.

These differences arise from the basic hypotheses needed by the procedures. In fact, from this viewpoint, the two procedures are very different. Moenck's procedure yields solutions for certain kinds of summands. In contrast, Gosper's procedure yields solutions when the sought closed form satisfies certain conditions. Thus, Gosper's hypotheses are based on the form of the solution to the problem.

Another difference between the two procedures is that, when applying Gosper's procedure, one does not know, by just looking at the summand, whether there will be a closed form or not. One has to carry out the whole procedure to know this. Moreover, while executing Moenck's algorithm, one can know whether the result will be a definable closed form when one determines whether there exists a transcendental part or not.

5.2.3. Karr's Procedure

This procedure is different in its approach from the previous two. It uses a field-theoretic motivation. It poses the problem in an algebraic way, and then derives conditions for summability. Starting with a field of constants, larger fields are constructed by the formal adjunction of symbols which behave like solutions to first order linear equations. Then, in these extension fields, the difference equations are posed and solutions are sought. The final criterion for summability is whether a certain algebraic extension is exactly of degree two.

Given a set of formal symbols and a sum, the procedure will determine when the sum is a rational function of the given symbols. For example, it will yield that $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is not a rational function of n. Its algebraic nature allows us to find formulae, as rational functions of n and the symbol H_n, for

$$\sum_{i=1}^{n} H_i \quad \text{and} \quad \sum_{i=1}^{n} iH_i \ .$$

However, it is not possible to handle sums in which one of the limits also appears in the summand.

Thus, none of the three methods is uniformly more powerful than the other two. The only way we may obtain an expression for H_n is with Moenck's procedure. Gosper's procedure does handle all the elementary sums that Karr's procedure does, but Karr's procedure provides a way of dealing with sums of the kind $\sum_{i=1}^{n} iH_i$, which are not handled by Gosper's procedure. On the other hand, Gosper's procedure is able to work with certain sums in which the summation limit appears in the summand.

The basis of Karr's algorithm are the *difference fields*. Their definition is quite elegant: A difference field is a field F together with an automorphism σ of F. Finding a sum in "finite terms" can now be described as: given f, and a difference field F of which f is an element, we look for solutions g of

$$\sigma g - g = f$$

only in F. Thus, the choice of field F is the means by which "in finite terms" is given a precise meaning.

To present a complete sketch of the procedure one needs to introduce and develop several concepts which would require many pages. They can all be found in [Kar79]. The essence of the procedure, however, is that one may symbolically extend the starting field and check, algorithmically, that certain necessary conditions hold. Once this is accomplished, a vector space basis is explicitly computed and its expression yields the desired closed form.

Knowing how to compute answers to questions about the homogeneous group and solutions for first order linear difference equations in a $\Pi\Sigma$-field F, Karr's procedure

(1) algorithmically determines, given $\alpha, \beta \in F$, whether F(t) with $\sigma t = \alpha t + \beta$ is a $\Pi\Sigma$-extension of F

(2) is able to lift to F(t) the computations concerning the homogeneous group and difference equations if F(t) described in (1) is indeed a $\Pi\Sigma$-extension of F.

The construction starts with a constant field, in which certain computations are possible, and is guided solely by the successive choices of the pairs α, β from ever longer fields.

It seems plausible, although we have not been able to prove it, that there should be an algebraic method in the spirit of Karr's which would totally include Gosper's procedure. This conjecture is based on the striking analogies that exist between the two approaches. After all, the automorphism σ in the difference algebra is an abstraction of the shift operator.

5.3. Table-Driven Methods for Finding Closed Forms

These are procedures which, after modifying the data into appropriate forms, require a look-up facility to proceed. We shall briefly indicate two papers which have dealt with the problem in this way.

Cohen and Katcoff [Coh77], present a program which finds solutions for some systems of linear finite difference equations. The method used is a modification of the second method presented in Section 5.1. The authors show that it is possible to express the solution of a linear difference equation with constant coefficients as a formula which contains a finite summation. This allows them to delay the look-up of generating functions to a later stage. Another characteristic of this approach is that, if the program cannot find a closed form for the resulting summation, a formula with the sum displayed symbolically can be made available to the user. The user may then provide the system with a closed form.

Ivie [Ivi78], presents some programs written in the MACSYMA programming language which find closed forms for single variable recurrence relations with constant coefficients as well as some variable coefficient recurrences. As systems of recurrence relations with constant coefficients may be reduced to a sequence of single variable recurrence relations with constant coefficients, Ivie's programs may also be used in this more general setting.

The constant coefficient case is treated in the standard way, making extensive use of MACSYMA's facilities. One very interesting aspect of this paper is the approach taken to deal with the variable coefficient case. The method of exponential generating functions Y(x) is used. It is shown that an Ordinary Differential Equation, ODE, is obtained for Y(x). Thus, the method converts the solution of a recurrence relation to the solution of a differential equation. Of all the procedures we have seen, this is the only instance when such a reduction is used. Using the MACSYMA commands ODE2 and POWERSERIES, this method can be programmed. It is then left to the abilities of the system to find the desired solutions.

5.4. On Iterations With Multiple Inner Paths

So far we have been concerned with the problem of finding closed forms in the context of iterations with an irreducible body. We shall now analyze the case when there are alternations within an iteration L.

The presence of alternations within an iteration implies that there are multiple paths which the flow of control may follow once the T-branch of the iteration L has been taken. The relevance of this is that control variables may be modified differently in each existing path. So, the task of finding definable iterations becomes more complex.

Throughout this section, we assume that the iteration $L = <B, \varphi, \alpha, \beta>$ has n alternations and no iteration within its body B. It is easy to see that, under these conditions, there will exist at least n+1 and at most 2^n paths. Let $P_1, \dots, P_m$ be an enumeration of the paths within L where $n+1 \le m \le 2^n$. Let us say that ψ_i is the predicate which, when true, cause the flow of control to traverse P_i. The predicate ψ_i can be obtained by the conjunction of all the predicates whose truth determines a given path within the iteration. Moreover, let B_i be the sequence of statements executed in the path P_i, called the body of P_r. Figure 5.4.1 depicts an example of an iteration with six internal paths and Table 5.4.1 lists the associated information. As we assume no iterations exist within B, all bodies B_i may be viewed as basic blocks of instructions.

In Example 2.4.1 we presented an alternation with two inner paths in which the selection of branch in the alternation was dependent exclusively on the input data. Thus, the traversal order of the distinct paths could be made arbitrary by an appropriate choice of values for the input variables. It is not

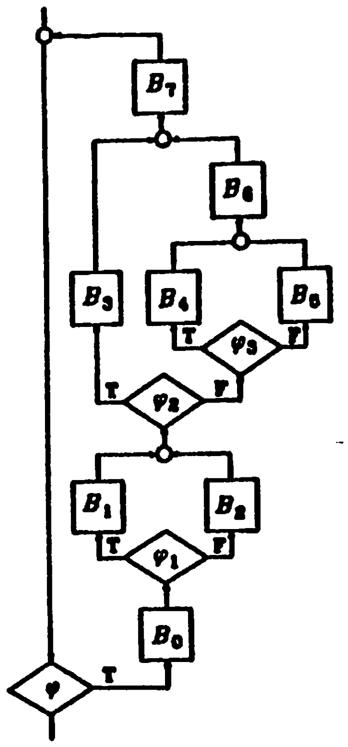

Figure 5.4.1

path	predicate ψ_i	body
P_1	φ_1 & φ_2	$B_0B_1B_3B_7$
P_2	$\neg\varphi_1$ & φ_2	$B_0B_2B_3B_7$
P_3	φ_1 & $\neg\varphi_2$ & φ_3	$B_0B_1B_4B_6B_7$
P_4	$\neg\varphi_1$ & $\neg\varphi_2$ & φ_3	$B_0B_2B_4B_6B_7$
P_5	φ_1 & $\neg\varphi_2$ & $\neg\varphi_3$	$B_0B_1B_5B_6B_7$
P_6	$\neg\varphi_1$ & $\neg\varphi_2$ & $\neg\varphi_3$	$B_0B_2B_5B_6B_7$

Table 5.4.1 Paths of Figure 5.4.1

difficult to generalize this example to one where there are m paths within an iteration and such that the flow of control will successively traverse branches in any predetermined order. As these cases are quite hopeless, and the skeleton approach is the only viable alternative to follow, they will not worry

us. We shall concentrate on cases when one can apply either the techniques of the previous sections or the ones to be introduced in this section.

The strongest condition which may occur is that, for a given run, only one path P_i is traversed each time the T-branch of L is traversed, i.e., that the hypotheses of Theorem 2.4.5 are satisfied. We thus obtain

Theorem 5.4.1

Let L = <B, φ, α, β> be an iteration with n alternations and no iterations within its body B. Assume that these n alternations determine m distinct paths within B, each with body B_i, and that ψ_i is the predicate which, when true, will determine the traversal of path B_i. If all the alternations within L are well behaved (see Theorem 2.4.5), then, for any variable x, a closed form for x exists in L iff there exist closed forms for x in each iteration <B_i, φ & ψ_i, α, β>, where $1 \le i \le m$.

Proof

If we have a closed form τ for x in L, then the special term τ will induce a closed form for x for each iteration <B_i, φ & ψ_i, α, β>. Namely, the one which is used in runs where the predicate φ & ψ_i is true every time the T-branch of L is taken.

As for the "only if" part, assume that τ_1, ... , τ_m, are the closed forms for x in <B_1, φ & ψ_1, α, β>, ... , <B_m, φ & ψ_m, α, β> respectively. Then, a closed form τ for x in L can be expressed by the special term

IF φ & ψ_1, 1 THEN τ_1 ELSE IF φ & ψ_2, 1 THEN τ_2 ELSE ... IF φ & ψ_m, 1 THEN τ_m ELSE A FI ... FI FI .

∎

Corollary 5.4.2

Under the same hypotheses of Theorem 5.4.1, τ is a definable closed form iff all the τ_i are definable closed forms.

Proof

If τ has no n-place function symbols f, then none of the τ_i constructed from it will contain any such symbol. On the other hand, if none of the τ_i has n-place function symbols f, then the construction of τ we exhibited in the proof of Theorem 5.4.1 has no n-place function symbols f.

■

As in Section 2.4, we obtain the following result about the predicates φ & ψ_i:

Corollary 5.4.3

The alternations within L are well behaved iff for all assignment functions ι, and all pairs $j_1, j_2 \le m$, where $j_1 \ne j_2$,

$$((\varphi \rightarrow \psi_{j_1}) \ \& \ (\varphi \rightarrow \psi_{j_2}))[\iota]$$

is false.

Proof

Like that of Theorem 2.4.6. The condition stated on the predicates precludes the possibility that two distinct branches be traversed in a given run.

■

When these conditions are not met, we know that our performance representations are not optimum. Some kind of iterative procedure must be used to correctly represent the profile equations of the program. However, there are two situations in which one can expect to outperform the skeleton approach. The first is based on a condition on the predicates and the second on a condition on the actions of the paths on the variables.

Theorem 5.4.4

If all predicates ψ_j are stable (see Def. 3.2.1), and there exist closed forms for all control variables in all paths, then the following algorithm outperforms the skeleton and should be used to find the action of the iteration on any variable x for which closed forms exist in all paths:

(i) determine which ψ_j is true, call it ψ_i;

(ii) update ψ_i's control variables, reevaluate, and record the number of times ψ_i evaluates to true until it becomes false;

(iii) when ψ_i becomes false, update the value of all control variables appearing in the other predicates using the closed forms and the current value of x;

(iv) go to (i).

Proof

By their definition, the predicates ψ_j are pairwise incompatible, i.e., ψ_j & ψ_k is never true, when $j \ne k$. Thus, at most one of them will be true at any given time. By the stability assumption, once ψ_j becomes false it will never become true again. (Stable is used in the same sense as in Def. 3.2.1 if one views the predicate ψ_j in the context of the iteration $<B, \varphi \,\&\, \psi_j, \alpha, \beta>$. Otherwise we redefine predicate stability, allowing the predicates ψ_j to have exactly two changes in their truth values). The savings in running time occur because we evaluate less times than in the skeleton different control variables, because once a predicate becomes false those control variables which only appear in it are never updated again, and because the variable x is updated less times than in the skeleton. The hypothesis of existence of closed forms is needed to perform the updates in an efficient way.

■

Corollary 5.4.5:

Under the hypotheses for L given in Theorem 5.4.1 and those of Theorem 5.4.4, if all the iterations $<B, \varphi \,\&\, \psi_j, \alpha, \beta>$ are definable, then $\#\varphi$, for $L = <B, \varphi, \alpha, \beta>$, can be obtained in linear time.

Proof

Using the algorithm described in Theorem 5.4.4. Once we discover which predicate is true, because of the definability of the corresponding iteration we only need to do one evaluation to find the number of consecutive times that the branch is to be traversed, and one update of all variables to reflect the consecutive traversals. This is done exactly m times, and the cost is linear in the number of paths.

■

So, when we have definability of $<B, \varphi \,\&\, \psi_j, \alpha, \beta>$ and stability of $\varphi \,\&\, \psi_j$, we obtain that $\#\varphi$ can be computed in linear time. This result can be exploited to avoid using the full skeleton when no closed forms can be found for a variable of interest which is modified in L .

We shall use the symbol $\circ$ to denote function composition. Given a variable x and two bodies B_i and B_j, we say that the actions f of B_i and g of B_j on x commute, if $f\circ g = g\circ f$ (i.e., the order in which the evaluation is performed is irrelevant). The commutativity hypothesis has the property that it can be checked by an algebraic manipulation program.

Theorem 5.4.6

Let $L = <B, \varphi, \alpha, \beta>$ be an iteration with n alternations and no iterations within its body B. Assume that these n alternations determine m distinct paths within B, each with body B_j, and that ψ_i is the predicate which, when true, will determine that path B_i is traversed. Assume further that there exist closed forms for all actions on x, and that all the actions on x commute. Then, $x[\#\varphi[i]]$ can be defined by a sequence of m evaluations.

Proof

Consider the following method:

(i) run the skeleton of L;

(ii) update x using the closed form for it in each branch B_i, $1 \leq i \leq m$.

That the correct value for $x[\#\varphi[i]]$ is obtained is a consequence of the commutativity assumption.

■

Theorem 5.4.7

Under the hypotheses of Theorem 5.4.6, if all the iterations $<B, \varphi \,\&\, \varphi_q, \alpha, \beta>$ are definable, then $x[\#\varphi[i]]$ may be obtained without using the skeleton for L .

Proof

The algorithm that allows us to do this is the one described in Theorem 3.1.3 for predicates that were a disjunction of other predicates. We only need to notice that L will be traversed as long as $(\varphi \,\&\, \psi_1)$ OR $(\varphi \,\&\, \psi_2)$ OR $\cdots$ OR $(\varphi \,\&\, \psi_n)$ is true.

■

In [Che78] conditional recurrence relations were introduced. They were formulated as recurrence relations of the form

$A_{k+1}(x) =$ case

$\varphi_1(k, x) \rightarrow f_1(k, x, A_k(h_1(x, k)))$

. . .

$\varphi_n(k, x) \rightarrow f_n(k, x, \Lambda_k(h_n(x, k)))$

end

where no two predicates φ_i could be simultaneously true. The method proposed for solving them was algorithmic in nature and required, as we do, closed forms for each "arm" of the recurrence to exist. Moreover, the only time they could assure an answer was when a special case of our hypothesis of stability for all predicates φ_i held.

In essence, the way one attempts to find a (conditional) closed form which is the solution to a conditional recurrence relation is by reducing the problem to that of finding a sequence of closed forms for the component recurrence relations, where the boundary conditions depend on the solution of a previous arm of the recurrence relation.

There is one particular case of iterations with multiple inner paths which is rather common and reduces very nicely to the standard case. Consider an iteration L with only two paths in it. That is, there exists only one alternation within the iteration. Let these two paths be P_1 and P_2, and assume further that every time the flow of control traverses L's T-branch for the first N times, where N is fixed, branch P_1 is executed. All other traversals of L's T-branch result in the execution of branch P_2. Then the conditional recurrence relation for L can be expressed by the following two recurrence relations:

$$<x[0], x[k], x[k+1]> \quad \text{for} \quad 1 \le k \le N$$

$$<x[N], x[k], x[k+1]> \quad \text{for} \quad k > N \quad .$$

The first of these relations will describe the action that P_1 effects on x and the second that of P_2 on x .

Two instances of this case can be seen in our Example 1.2.1, where N is equal to 1. In both of these cases, branch P_1 prints a headline and branch P_2 computes and prints the values of a table.

5.5. On Nested Iterations

We shall now study the problem of the existence of closed forms in the presence of nested iterations. The symbolic action of one iteration on a variable can be expressed as follows: let $\tau(x, k)$ be a closed form for x in L = <B, φ, α, β>. Then, for any execution of the program, the expression for x at β is the special term

$$\text{IF } \varphi.\ 1 \text{ THEN } \tau(x,\ \#\varphi(\hat{x}[\alpha])) \text{ ELSE } x \text{ FI} \quad .$$

The effect of multiple nested iterations on variables becomes quite involved. We shall illustrate this by presenting the case of two nested iterations and using Figure 5.5.1 as reference. Given a variable x, we let $x[k, \gamma_i]$.

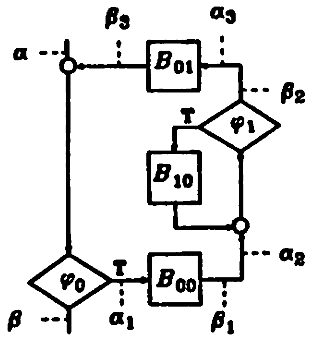

Figure 5.5.1

where $\gamma \in \{\alpha, \beta\}$, denote the value that x has at the point γ_i after traversing the corresponding iteration k consecutive times. We shall assume that, for $i \in \{0, 1\}$, the entries of vector $\mathbf{x}_i$ are φ_i's control variables. Moreover, for $i,j \in \{0, 1\}$, the action of B_{ij} on $\mathbf{x}_h$ is the (vector) function F^h_{ij}, and that on x is the function g_{ij}.

With this notation we may derive the expression for $x[k, \gamma_i]$, where $\gamma \in \{\alpha, \beta\}$ and $i \in \{0, 1\}$. We let $T_0(\mathbf{x}_0, k)$, $T_1(\mathbf{x}_1, k)$ be vectors of closed forms for $\mathbf{x}_0$ and $\mathbf{x}_1$ in the inner iteration, and $\tau(x, k)$ be a closed form for x in the inner iteration. The three basic relationships for x are the following:

(1) $x[k+1, \beta_1] = g_{00}(x[k, \beta_2])$

(2) $x[k+1, \beta_2] = \tau_1(x[k+1, \alpha_2], \#\varphi_1(\mathbf{x}_1[k+1, \alpha_2]))$

(3) $x[k+1, \beta_3] = g_{01}(x[k+1, \alpha_3])$.

When $k = 0$, relationships (1) becomes: $x[1, \beta_1] = g_{00}(x[\alpha])$.

From (2) we realize that, to express the effect of a nested pair of iterations on a variable symbolically, one needs to describe the effect of successive entrances to the inner iteration. We shall now introduce some notation which will aid us in this description for the vectors of control variables $\mathbf{x}_0$ and $\mathbf{x}_1$. We define the functions T_0 and T_1 as follows:

Definition 5.5.1

For $i \in \{0, 1\}$,

$T^0_i(\mathbf{x}_0, \mathbf{x}_i) = \mathbf{x}_i$

$T^{\ell+1}_i(\mathbf{x}_0, \mathbf{x}_i) = F^i_{01} \circ T_i(F^i_{00} \circ T^\ell_i(\mathbf{x}_0, \mathbf{x}_i), \#\varphi_1(F^1_{00} \circ T^\ell_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha]))$

■

Theorem 5.5.1

(i) The $(\ell+1)^{st}$ consecutive time we enter the outer iteration's T-branch, the inner iteration is traversed

$$\#\varphi_1\left(F^1_{00} \circ T^\ell_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha]\right)$$

times.

(ii) The value of $\mathbf{x}_i$ at β_2 after traversing consecutively the T-branch of the inner iteration ℓ times is:

$$\mathbf{x}_i[\ell, \beta_3] = T^\ell_i(\mathbf{x}_0, \mathbf{x}_i)[\alpha] \quad .$$

Proof

By induction on ℓ, we prove simultaneously (i) and (ii). For $\ell = 1$ we have that

$$\mathbf{x}_1[1, \alpha_2] = F^1_{00}(\mathbf{x}_1[\alpha]) = F^1_{00} \circ T^0_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha],$$

and so the inner iteration is traversed $\#\varphi_1(F^1_{00} \circ T^0_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha])$ times. Thus, (i) holds for $\ell = 1$. Because of this, we have that

$$\mathbf{x}_i[1, \beta_3] = F^i_{01} \circ T_i((F^i_{00} \circ T^0_i(\mathbf{x}_0, \mathbf{x}_i)[\alpha]), \#\varphi_1(F^1_{00} \circ T^0_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha]))$$

and (ii) holds for $\ell = 1$.

Assume now that (i) and (ii) hold for ℓ. Then, as $\mathbf{x}_i[\ell+1, \alpha_1] = \mathbf{x}_i[\ell, \beta_3]$, we have that

$$\mathbf{x}_i[\ell+1, \alpha_2] = F^i_{00} \circ T^\ell_i(\mathbf{x}_0, \mathbf{x}_i)[\alpha] ,$$

the inner iteration is traversed

$$\#\varphi_1(F^1_{00} \circ T^\ell_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha])$$

times, and (i) holds for $\ell+1$. As for (ii), using (i) we have that

$$\begin{aligned} \mathbf{x}_i[\ell+1, \beta_2] &= T_i(\mathbf{x}_i[\ell+1, \alpha_2], \#\varphi_1(F^1_{00} \circ T^\ell_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha])) \\ &= T_i(F^i_{00} \circ T^\ell_i(\mathbf{x}_0, \mathbf{x}_i)[\alpha], \#\varphi_1(F^1_{00} \circ T^\ell_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha])) \end{aligned}$$

and thus

$$\mathbf{x}_i[\ell+1, \beta_3] = F^i_{01} \circ T_i(F^i_{00} \circ T^\ell_i(\mathbf{x}_0, \mathbf{x}_i)[\alpha], \#\varphi_1(F^1_{00} \circ T^\ell_1(\mathbf{x}_0, \mathbf{x}_1)[\alpha]))$$

$$= T_l^{l+1}(\mathfrak{x}_0, \mathfrak{x}_1)[\alpha] .$$

■

In complete analogy with T_n, we define v_n to describe the effect of the nested pair of iterations in Figure 5.5.1 on a variable x.

Definition 5.5.2

$v^0(x) = x$

$v^{n+1}(x) = g_{01} \circ \tau(g_{00} \circ v^n(x), \# \varphi_1(F_{00}^1 \circ T_1^n(\mathfrak{x}_0, \mathfrak{x}_1)[\alpha])) .$

■

Theorem 5.5.2:

The value of x at β_3, after traversing consecutively the T-branch of the inner iteration l times, is:

$$x[l, \beta_3] = v^l(x)[\alpha] .$$

Proof

Analogous to that of Theorem 5.5.1, but using Definition 5.5.2.

■

We are only now able to express $x[\beta]$, namely,

$$x[\beta] = x[\# \varphi_0(\mathfrak{x}_0[\alpha]), \beta_3] = v^{\# \varphi_0(\mathfrak{x}_0[\alpha])}(x)[\alpha] .$$

Thus, the existence of closed forms for this case requires that both iterations be definable and that the functions T^n and v^n have closed forms.

In our Examples 1.2.2 and 1.2.3, as well as in all the examples mentioning nested iterations that we have seen in the literature, the inner iteration is always traversed a fixed number of times, say N, each time the T-branch of the outer iteration is traversed. This is the same condition we require in Theorem 2.4.5 to obtain an optimal profile representation. Under these conditions, $v^l(x)$ has a simple expression:

$$v^{l+1}(x) = g_{01} \circ \tau\left(g_{00} \circ v^l(x), N\right) .$$

When $l = 3$, this can be fully expanded to

$$v^3(x) = g_{01} \circ \tau\left(g_{00} \circ g_{01} \circ \tau\left(g_{00} \circ g_{01} \circ \tau\left(g_{00}(x), N\right), N\right), N\right) .$$

So, when v^l has the above simple expression, we see that $v^l(x)$ can be obtained by $3l$ symbolic evaluations, i.e., the complexity of finding the expression for $x[\beta]$ depends linearly on the values of $\# \varphi_0$. The existence of a closed form $\tau(x, k)$ for x in the inner iteration is all we need to achieve this.

If $g_{00} \circ g_{01}$ is the identity function, then

$$v^{l+1}(x) = g_{01} \circ \tau\left(g_{00}(x), N^{l+1}\right) .$$

Then, in this case we obtain only from τ a closed form for x in the system of two nested iterations. We have thus proven

Theorem 5.5.3

Consider the pair of nested iterations as depicted in Figure 5.5.1, and assume that $\tau(x, k)$ is a closed form for x for the inner iteration. Then

(1) if the inner iteration is well behaved (see Theorem 2.4.5), $v^l(x)$ can be obtained by $3l$ expression evaluations;

(2) if we also assume that $g_{00} \circ g_{01}$ is the identity function, then $v^l(x)$ has a closed form.

■

5.6. Summary

In this chapter we have dealt with the problem of finding closed forms for the actions of programs on variables. It was seen that, under special circumstances, finding closed forms can be posed as a recurrence relation problem. Three known automatic procedures for finding closed forms for special kinds of recurrences were presented. Table look-up methods for solving recurrences were also explained. Finally, the cases of iterations with multiple inner paths and of nested iterations were analyzed and expressions for their (combined) action on variables were obtained.

CHAPTER 6

On Recursive Programs

So far we have not dealt with the problem of recursion. In fact we have not specified how to treat procedures or subroutines. We have viewed programs as syntactic objects in which each statement has a well defined effect on the value of variables, but we have not been concerned with the various possible types of statements.

6.1. Procedures as Basic Blocks

When using the D-chart representation of programs, basic blocks have been assumed to contain primitive statements, so that a term of our performance language exists which describes the action of the block on the variables. This is essentially equivalent to assuming that, at the point of a procedure call, one performs "in-line code expansion", and replaces the call by the body of the procedure. In a D-chart representation, the procedure call statement corresponds to a node which is then expanded to the full D-chart representing the procedure body.

Our techniques already allow us to do better than in-line code expansion. We can derive the profile of each procedure and use it to obtain the profile of the whole program. Our methods allow us to treat nonrecursive procedures as self-contained units. The skeleton approach can be used if the procedure does not satisfy the necessary definability conditions needed for a faster representation.

When dealing with specific programming languages, care has to be taken in determining the meaning of input and output variables in a procedure. Problems with aliasing, sharing and external variables have to be addressed. To give just one example, the variables in FORTRAN COMMON statements can play the role of input variables, output variables, or both. This analysis is context dependent but can be carried out automatically by existing techniques of compiler theory.

Cheatham states that the analysis of a procedure should not be made in total isolation [Che79], since all the possible patterns of sharing among formal parameters and input variables have to be explored. The argument supporting this statement is that, in the context of a program, a procedure is normally called making use not of all but of a few of the existing possibilities. When the programming language allows for a wide variety of combinations, this suggestion seems appropriate. The alternative method proposed in [Che79] for analyzing procedures was based on a case-at-the-time approach.

When a call to an as yet unanalyzed procedure is found, the procedure is analyzed in the environment of the call. A *template*, giving a generalized description of the call environment, is created and kept in a library. This template contains the modes of the actual parameter values and input variables, and the sharing patterns among them. Then the procedure is analyzed assuming this (restricted) environment and the results stored associated with the corresponding template. When subsequent calls to the same procedure are encountered, the new call environment is compared with the existing templates, and, if a match is found, the previous analysis is used. If the modes of the actual parameter values and input variables do not coincide or if the sharing patterns are distinct, then a new template is created, added

to the library, and the corresponding analysis is performed and stored as well.

This method permits the treatment of fairly general cases. In the context of programming languages with strict sharing and aliasing rules, or when a procedure does not use these features of the language, the method reduces to that which performs the analysis in complete isolation.

Thus, the problem of nonrecursive procedures can be treated with the techniques already developed in the previous chapters. Unlike the METRIC system [Weg75], this approach need not make any assumptions on the order and on the sequence of procedure calls.

6.2. Recursive Procedures

There are several reasons why the analysis of recursive procedures is more complex than that of nonrecursive ones. For example, it is proven in Manna's book on the Mathematical Theory of Computation, [Man74], that any flowchart schema (e.g., a D-chart) can be translated into an equivalent recursive schema. Thus, the class of computations which can be described with D-charts is included in the class of computations which can be described using recursive schemas.

Recursive schemas can be defined as follows. Consider the language introduced in Chapter 2 with no special denotation symbols nor the special symbol IFTHENELSEFI. Then a *recursive* schema σ over the set of formulae in the language is of the form:

$\mathbf{z} = \tau_0(\mathbf{x}, \mathbf{F})$, where

$\mathbf{F}_1(\mathbf{x}, \mathbf{y}) := \tau_1(\mathbf{x}, \mathbf{y}, \mathbf{F})$

. . .

$\mathcal{f}_N(\mathfrak{x}, \mathfrak{y}) := \tau_N(\mathfrak{x}, \mathfrak{y}, \mathcal{f})$

Here $\tau_0(\mathfrak{x}, \mathcal{f})$ is a predicate that contains no variables other than the input variables $\mathfrak{x}$ and the function variables $\mathcal{f}$. Similarly, $\tau_i(\mathfrak{x}, \mathfrak{y}, \mathcal{f})$, $1 \le i \le N$, is a predicate that contains no variables other than the input variables $\mathfrak{x}$, control (noninput) variables $\mathfrak{y}$, and the function variables $\mathcal{f}$.

The following theorem is stated in [Man74], page 328:

Theorem 6.2.1 **(Patterson, Hewitt)**

There is no flowchart schema (with any number of program variables) which is equivalent to the recursive schema

S_A $z = F(a)$ where

$F(y) :=$ if $p(y)$ then $f(y)$ else $h(F(g_1(y)), F(g_2(y)))$

∎

Thus, recursion is seen to be more powerful than iteration. It is worth noting that it is essential that both F's occur as arguments of h in the definition of S_A, because there are flowchart schemas which are equivalent to the following two recursive schemas:

S_B $z = F(a)$ where

$F(y) :=$ if $p(y)$ then $f(y)$ else $h(F(g(y)))$

and

S_C $z = F(a)$ where

$F(y) :=$ if $p(y)$ then $f(y)$ else $h(y, F(g(y)))$.

Other issues which make the analysis of recursive procedures more complicated, in the context of symbolic evaluation, arise from the fact that

procedures in general may return addresses as values. Our techniques have not dealt with this problem, because in normal program statements it is the value of the variable which is passed and modified.

There are two situations in which one may transform the situation from passing an address back to passing a value. Consider a recursive procedure F with only one argument, say x:

(1) If the formal parameter x is bound by value, then on each recursive call a new copy of the parameter is created. When "unwinding" the nested calls, appropriate care has to be taken to use the correct value of x at each point of the execution path. One essentially mimics the process of performing a procedure call and preserving in a stack the appropriate information.

(2) If x is bound by reference (and always occurs in the same position as an actual argument in the recursive call), then the effects on x during the recursive descent and unwind accumulate much like what happens to the variables in a regular iteration.

These considerations can be easily generalized to the case of several formal parameters.

The body of a recursive procedure may contain several paths where recursive calls are made and others which are recursion free. Moreover, in a given path, there could be several recursive calls. If the pattern of calls becomes too involved, or if definability conditions do not exist, an extension of the skeleton approach can be used to produce the desired profile. There are essentially no new problems in obtaining skeleton representations for recursive programs.

However, there are several cases when one can expect to do better than using the skeleton. Two cases which have been mentioned in the literature, although in settings which were more restrictive than ours, are the following:

(1) Wegbreit requires [Weg75], in order to convert to a certain "normal form", that the procedures be *well nested*, i.e., that, whenever A calls B, no procedure called by B calls A.

(2) Cheatham [Che79] analyzes the case of *simple recursive* procedures, i.e., those which have at most one recursive call along any path (including iterations) from its entry to an exit.

When either of these hypotheses hold, our methods developed in Chapters 2, 3 and 5 are applicable. In fact, (1) allows us to transform the procedure into one which essentially looks like an iteration with multiple inner paths. Then, our results of Section 5.4 can be directly applied.

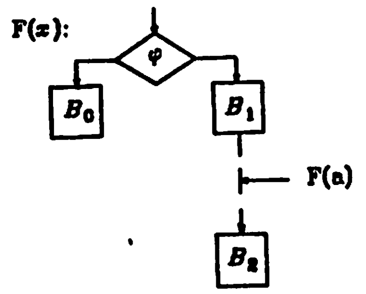

Figure 6.2.1

As for (2), consider the procedure $F(x)$ represented in Figure 6.2.1, where the arrow pointing to the broken line between B_1 and B_2 means that a call to F with formal parameter a is made at that point. Viewed in this way, the behavior of F can be thought of as an iteration with body B_1 of recursive descents, a computation B_0 when control has reached "bottom", and then an iteration B_2 to "unwind" the recursive descent. In the iteration B_1, the initial value of parameter x will be the value of the actual parameter, and the "next traversal" value will be a. In the iteration B_2, the initial value will be that computed in B_0, and the next traversal value will be that computed in B_2. The number of traversals taken for each iteration is determined when the predicate φ causes B_0 to be executed instead of B_1. We may then employ the techniques developed in Chapters 2 through 4, to analyze the pair of iterations in order to determine the effects of a call on F.

6.3. Two Examples

In this section we shall analyze two examples of recursive procedures which require distinct kinds of performance representations. We shall first present a case where our techniques are directly applicable.

Example 6.3.1

Consider the recursive definition of the factorial function:

```
Factorial(n)
if n ≤ 1 then return(1);
else return(n × Factorial(n-1));
end
```

Using Figure 6.2.1 as reference, we have that φ is the predicate $n \le 1$, in B_0 we return the value 1, B_1 contains the statement $a := n - 1$, and in B_2 we multiply the current index by the value associated with the next index.

The control variable is n, and in the F-branch the only modification made to it is to decrease it by one in B_1. Thus, we are back to the case of linear function actions. We then know that the F-branch will be executed (n-1) times.

■

In this example, we can therefore build a ppf to represent the profile equations. We can do this because the transformations of the control variables are amenable to our previous analysis.

Example 6.3.2

The following algorithm will list, in pre-order, the names of the nodes in a binary tree:

```
Search-a-tree(pointer to the root)
        if pointer does not point to an empty tree
        then Search-a-tree(left pointer of root)
                write out the name of the root
                Search-a-tree(right pointer of root)
end
```

Our only tool to deal with this kind of algorithm, as no algebraic properties can be extracted from the usage of the control variables, is to represent it using the skeleton approach.

■

6.4. Summary

In this chapter we have seen how to treat procedures, and in particular recursive procedures. Even though recursion has been seen to be more powerful than iteration, the skeleton method can always be used to obtain performance representations. Moreover, there are several situations in which our techniques for nonrecursive programs are directly applicable and can improve on the skeleton approach. The restrictions which need to be satisfied are related to the kinds of parameter passing mechanisms used,

and to the complexity of the structure of procedure calls.

CHAPTER 7

A System for the Microanalysis of Programs

In this chapter we shall discuss some of the issues to be resolved when designing a system which implements the approach developed in the previous chapters. As specified in Chapter 1, our goal is, given a program, to build a performance representation corresponding to its profile equations. Moreover, this performance representation should yield the profile (much) faster than if the profile were obtained from a properly instrumented version of the original program. We shall call *microanalysis of a program* the process of finding, as a function of the input values, the exact profile of the program.

In Section 1.2 we introduced a default method: the skeleton. After showing why we should try to do better (see Examples 1.2.2 and 1.2.3), in Chapters 2 and 3 we developed an approach whose goal was to "linearize" loops. In Chapter 4 we showed how this could be achieved when the actions on the control variables were linear. Then Chapter 5 gave us tools to deal with more general situations.

We shall now discuss several aspects that we have not addressed before and propose a way in which all of our considerations could be implemented.

7.1. Non Linear Actions of Iterations on Variables

Already in Chapter 2 we recognized that our main problem was to analyze the effect that iterations have on variables. To determine $\#\varphi$, for a given iteration $L = \langle B, \varphi, \alpha, \beta \rangle$, and to obtain the action of an iteration on a variable in an efficient way, we needed closed forms.

In Chapter 4 we dealt with the case of control variables x which were modified by a traversal independent linear action, i.e.: $x[k+1] = ax[k] + b$, where a and b were not modified in the body of the iteration L. The procedures presented in Section 5.2 allowed us to deal with some instances of the transformation $x[k+1] = x[k] + f(k)$. Our discussion in Section 5.1 extended this to transformations of the type $x[k+1] = g(k)x[k] + f(k)$, but the evaluation cost was no longer linear in the length of the program.

The rest of Chapter 5 provided us with some more tools to find closed forms. However, there are many kinds of transformations which have not been dealt with. We shall now analyze some of them.

Nonlinear Traversal-Independent Actions

As noted in Section 5.1, when the relationship between $x[k+1]$ and $x[k]$ does not depend on the iteration index k, one has that

$$x[k+1] = g^k(x[0])$$

where g is the action of B on x. Thus finding (definable) closed forms is equivalent to finding a term $\tau(x,k)$ (with no n-place function symbols f) which describes $g^k(x)$. However, to have a definable iteration, one also needs to solve for k. This last condition was what presented all the problems in our analysis of the linear function case in Chapter 4.

There is one family of transformations which satisfies both of the constraints outlined above and which we have not dealt with. It is a transformation of multiplicative type: the action of B on x being $a^s x^r$, where a, s and r do not change in B. Then

$$x[k+1] = a^s(x[k])^r = a^{k(s+r)}(x[0])^{(k+1)r} \ .$$

The solution for k to the equation $x[k] = N$ is given by

$$k = \frac{\log(N) + (r+s)\log(a)}{(r+s)\log(a) + r\log(z[0])}$$

We notice that no hypotheses have been made about a, s and r. However, our ability to find closed forms seems to stop here with this kind of transformations. Even when we consider the action of B on a variable z to be $z^2 + b$, we do not have a non-procedural way to express $z[k+1]$. In fact, in this case we have $z[2] = (z[0]^2 + b)^2 + b$ and $z[3] = ((z[0]^2 + b)^2 + b)^2 + b$. The only expression for $z[k+1]$ we have been able to derive involves k nested sums. Transformations which are algebraically more complex do not lead to simplifications in the expressions they generate. Thus, when these kinds of actions are detected, one should proceed with the skeleton approach.

On Functions With Finite Range

Consider the case where $z[k+1] = z[k] + f(k)$ and the range of f is finite, i.e., there are only finitely many values produced by f. One example of this is given by $z[k+1] = z[k] + 1 + (k \bmod K)$, where K does not change in B. We shall see that, under some special circumstances, we may treat this case quite efficiently.

Functions f with finite range may be characterized as follows:

(I) Those functions, *periodic*, whose values determine a sequence of numbers composed of cycles, i.e.: there exists a positive number τ, called the period or cycle length, such that, for all numbers t, $g(t+\tau) = g(t)$.

(II) All others.

Naturally, functions of type (I) are amenable to a better (more efficient) analysis. We note in passing that $f(k) = 1 + (k \bmod K)$, where K is constant, is of type (I).

To analyze functions of type (I), let

$$S(j,\tau) = \sum_{i=j+1}^{\tau+j} f(i) \quad ,$$

where τ is f's period.

Theorem 7.1.1

For any two integers j_1 and j_2, $S(j_1, \tau) = S(j_2, \tau)$.

Proof

We shall show that, for all j, $S(j, \tau) = S(0, \tau)$.

Because of f's periodicity, $f(j) = f(j+\tau)$ for all integers j. So for any given integer j

$$\begin{aligned} S(0,\tau) &= \sum_{i=1}^{\tau} f(i) = \sum_{i=1}^{\tau} f(i+j) \\ &= \sum_{i=j+1}^{\tau+j} f(i) \\ &= S(j,\tau) \quad . \end{aligned}$$

∎

Let us denote $S(0, \tau)$ by $S(f)$.

With this notation we may easily describe the values a control variable z takes when a function f of finite cyclic range is acting on it in an additive way. We have that

$$z[k] = z[0] + \sum_{j=1}^{k-1} f(j) \, .$$

But this can be written as

$$z[k] = z[0] + \left\lfloor \frac{k}{\tau} \right\rfloor S(f) + \sum_{j=1}^{k \bmod \tau} f(j)$$

by just noting that, every τ traversals, $z[j+\tau] - z[j] = S(f)$. We also notice that

$$z[k+1] - z[k] = f(k+1) = f((k+1) \bmod \tau) \, .$$

If we consider

$$g(t) = \frac{S(f)}{\tau}t + x[0] \quad .$$

we have, for all integers n,

$$x[n\tau] = g(n\tau) \quad .$$

Thus, for the purposes of determining $\#\varphi$, we may use g as an approximation to f. We find the iteration index m at which one would stop if g were transforming the control variable, and then look at the interval of numbers $(n\tau, (n+1)\tau]$ in which m lies. In it we determine the exact iteration index for which L will halt.

Example 7.1.1

Let $L = \langle B, x \leq 2000, \alpha, \beta \rangle$, where $x[\alpha] = k[\alpha] = 0$, the only modification to x in B is $x := x + 1 + (k \bmod 20)$, and the only modification to k in B is $k := k+1$.

Then τ is 20 and $S(f) = \sum_{i=1}^{20}(1 + (i \bmod 20)) = \sum_{i=1}^{20} i = 210$. Thus $g(t) = \frac{210}{20}t$ and so the value for $\#\varphi$ lies in the interval (180, 210] (since $\left\lceil\frac{20\times2000}{210}\right\rceil = 191$).

Thus, we only need to find the least k > 180 such that $k = 9S(f) + \sum_{i=1}^{k_1} f(i)$ and $k > 2000$. The desired k_1 is easily seen to be 15. In fact, as we have $\sum_{i=1}^{k_1} i = \frac{k_1(k_1+1)}{2}$, one can determine k_1 analytically. Thus, $\#\varphi = 195$ when $x[\alpha] = k[\alpha] = 0$.

■

For functions of type (II), our skeleton approach seems to be the only generally applicable approach.

7.2. On Finding Roots at Run-time or by User Supplied Factorizations

In Chapter 3 we saw that the actions of iterations on variables may yield functions whose zeroes do not have an elementary expression. As these roots are intimately connected with counting functions $\#\varphi$ (see Theorem 3.2.1), determining these roots at run time might be a viable alternative.

The tradeoff one has to have in mind is the following: the execution time of the algorithm which finds the least nonnegative root of the function must be less than the running time of the skeleton of the corresponding iteration. In other words, one should not find roots at run-time if the process of doing so takes longer than that of performing the skeleton for the corresponding iteration.

However, there is another approach to this problem. Once the system has identified a function for which it does not know how to find zeroes symbolically, the user could be presented with it and asked for help. The user may then return a factored form, or a criterion, e.g.: apply a specific procedure for finding its roots, or perform a skeleton, or search a given database of function forms.

With this flexibility one can envision that improvements to a given performance representation could be achieved by studying and analyzing in detail the parts of this representation where the default procedure was used. As these studies could be done after a given representation has been created, one would benefit from having such a representation for performance studies while improving it. This interactive approach seems to us to be the most appropriate.

7.3. On Non-Numeric Control Variables

Throughout our discussion in Chapters 2 through 5, we have assumed that control variables are of type numeric. Moreover, we have made extensive use of the ordering and algebraic properties that numeric entities have. In fact, their algebraic properties have been most useful for our purposes. We have been able to "linearize loops" only when we obtain symbolic expressions for the roots of certain functions associated with the iterations.

The skeleton approach of Section 1.2 can be used for all types of control variables. To apply our methods which improve on the skeleton, however, we rely on an underlying algebraic structure. Thus, if one has a data type which has an underlying algebraic structure amenable to that of the numeric type, our methods can be used without problems. Otherwise, other techniques need to be developed.

For example, in several languages the type char (character) is declared as an enumerated type. In such cases it is often true that the letters in the alphabet, upper and lower case, are assigned a place in a linear ordering. Let us assume that we deal with a language where all lower case letters precede upper case ones, and that the ordering goes from a to z and from A to Z. Then, there may be situations where one may recapture properties of numbers. For example, it now may make sense to say that an iteration is going to be traversed Z - g times, by making use of the underlying order.

To determine these cases, each data type which we would like to treat efficiently must be analyzed individually. Those cases in which a better method than the skeleton can be used should be treated accordingly. It is our belief, however, that, if one is able to deal properly with iterations whose control variables are of type numeric, most of the achievable efficiency of a performance representation for computation intensive programs is achieved.

We have thus far not talked about the cost of building a performance representation, but it should be clear that the more "intelligence" we incorporate into the system which builds such a representation, the costlier it becomes. Of course, the expected return is that the performance representations produced by the system should execute faster.

There is one kind of control variables for which we do not think too many improvements over the skeleton can be obtained in an automatic way. This is the case of control variables of type boolean, i.e., which only take the values true or false when evaluated. The reason for our skepticism is that in the body of the iteration a (possibly very complicated) condition must be set equal to the variable, and thus too much information may be lost in this assignment. An alternative way to deal with this case is to try to obtain a symbolic expression for the variable and then use an algebraic manipulator system to check for the truth value. But even this is not entirely satisfactory because it does not help us find the number of consecutive times that the variable will evaluate to true.

In fact, trying to deal with boolean valued variables in an automatic way brings us into the field of automatic theorem proving, a task known to be generally undecidable. This is so because normal programming languages have enough algebraic expressive power to encode complex arithmetic statements whose truth may not be formally provable. Our next section deals with some issues related to this aspect.

7.4. The Automatic Verification of Hypotheses

What interests us is to establish in an automatic way the validity of the hypotheses needed to obtain our optimal performance representations. To do this, we have to assume some properties of the method used for finding the performance representations.

We assume that, after a first pass over the code whose performance representation is to be found, one has complete data flow analysis information for each variable at each point of the program. We also assume that a symbolic expression is obtained for each control variable in each basic block of statements. As we have remarked in Chapter 2, finding symbolic expressions for straight line code is not a problem. Determining which variables are control variables, however, does require the full power of data flow analysis techniques.

While parsing a program, it is easy to determine in an automatic way if an iteration has within its body other iterations or altèrnations. Thus, having detected an iteration with no other iterations nor alternations within it, and also knowing its control variables, the symbolic expressions for the control variables obtained by mere symbolic execution of straight line code yield a fair amount of information. Indeed, after reducing the symbolic expressions for the control variables to its simplest algebraic form, say by a system like MACSYMA, pattern matching techniques enable us to determine the algebraic complexity of the transformation as well as whether the transformation is traversal independent or not. We then act accordingly, using the techniques developed in Chapters 4 and 5.

If a closed form for some of the control variables cannot be found, then that iteration will have to be dealt with using the skeleton approach. If there exists a closed form for each control variable, we determine whether the iteration is definable. In case it is not definable, there still exists the option, as discussed in Section 7.2, of obtaining at run time roots for the function

determined by the closed forms of the control variables and of the predicate, instead of resorting to the skeleton approach. If one decides to obtain roots at run time, a routine for doing so will have to be incorporated into the performance representation at the appropriate point. Its inputs will be of two kinds: those whose values can be determined at compile time, and those whose values will only be obtained at run time.

The above analysis can be generalized to the case of iterations with several inner paths, which may include nested iterations as well. This is achieved with the help of the data flow information. Once we know that a certain variable is a control variable of an iteration, we may determine whether it is modified within an inner iteration or not, and also whether distinct branches modify it differently. If the control variables are not modified by inner iterations with a variable number of traversals, and if distinct inner branches produce the same effects on them, then one can carry out the same analysis as in the case of an irreducible iteration.

What we would like very much to determine in an automatic way is the validity of the hypotheses of Lemmas 2.4.3 and 2.4.4. For Lemma 2.4.3, the case of alternations within iterations, Theorem 2.4.6 is a rather suitable alternative. Unfortunately, we do not have anything similar for the iteration case. When using the condition presented in Theorem 2.4.6, that

$$((\varphi_0(\bar{x}_0) \to \varphi_1(\bar{x}_1)) \,\&\, (\varphi_0(\bar{x}_0) \to \neg\varphi_1(\bar{x}_1)))$$

be false under all assignment functions ι, for some families of predicates we may determine the validity of this condition in an automatic way. One instance of this is when the predicates are the atomic relational operators RO introduced in Chapter 4, and the variables involved are of type numeric. In the general case, however, one would need a universal theorem prover, which is known not to exist. We believe that this is another instance where the system should ask the user to decide how it should proceed.

For the case of nested iterations, there are some instances in which one can determine that the hypotheses of Lemma 2.4.4 hold. What we need is to establish that inner iterations are always traversed a fixed number of times each time the outer iteration is entered. This may be true because of the syntactic form of the construct, e.g., a FORTRAN DO statement where all bounds and the step do not change in that program segment, or because it can be determined from the first pass over the code of our procedure by using the data flow information. An example of the latter situation is when one finds a definable iteration where all the variables appearing in the expression of the counting function are not modified within the enclosing iteration. This was the case in Examples 1.2.2 and 1.2.3.

In Section 3.4 we have seen a general way to deal with the case when the hypotheses of Lemmas 2.4.3 and 2.4.4 are not true. This way consists of adopting an algorithmic ppf representation. However, it was pointed out that this approach may not be satisfactory. It may be that the skeleton runs faster than this representation. Two cases where the expected cost of running the algorithmic ppf is smaller than that of running the skeleton were discussed in Section 3.4:

(1) When we are given a definable iteration L with predicate $\varphi_0(\mathbf{x}_0)$, an alternation within this iteration with predicate $\varphi_1(\mathbf{x}_1)$, and closed form expressions for φ_1's control variables $\mathbf{x}_1$.

(2) When we are given a definable iteration L with predicate $\varphi_0(\mathbf{x}_0)$ and another definable iteration $\varphi_1(\mathbf{x}_1)$ within it, for whose control variables $\mathbf{x}_1$ we have closed form expressions.

The hypotheses for both of these cases can be established automatically as outlined in the previous paragraphs. Following the discussion of Section 5.4, (1) can be generalized to the case where several inner paths exist within the iteration.

There are several other hypotheses which a system may verify automatically. To mention just one more, in Section 5.4 we saw that, when the actions of two blocks on a variable commute, then we can improve on the skeleton even though definability may not exist. The commutativity hypothesis can be established automatically with the help, say, of an algebraic manipulation system like MACSYMA.

7.5. An Interactive System for Microanalyzing Programs

From our presentation it should be clear that an automatic system which would operate on programs and construct our "performance representations" can be implemented. There are several alternatives as to the organization of this software. A rather natural way of organizing it could be by separating the system into two parts:

(1) A "front end" subsystem that would parse, obtain all the data flow information needed to determine all control variables, and build an intermediate representation of the program.

(2) A "back end" subsystem that would operate on this intermediate form of programs and produce the final performance representation.

There are several advantages of this approach. One of them is that the front-end subsystem, which will necessarily have programming language dependent parts, can be modularly reprogrammed to perform the analysis of programs written in different programming languages. We thus avoid modifying the whole system when one wants to analyze programs in a new programming language. The back end would always act on a unique kind of intermediate representation. As outlined here, the front end can be a silent process, almost no interaction with the user being anticipated.

On the other hand, we believe that the back end should be interactive in nature to achieve better results. As we have mentioned in Sections 7.2 and 7.4, there are several instances where a dialogue with the user would help improve on the default procedure. Some instances of this are in assisting the system in finding closed forms, deciding on a numerical method for finding roots, providing the system with a factorization of a polynomial or helping the system establish the hypotheses needed to obtain optimal performance representations. It should also be noted that the whole process could be made totally silent because a skeleton, as described in Section 1.2, can be built with no more information than the text of the program to be analyzed.

As output, the back end would produce a performance representation in a specified programming language. One would then compile this new program and run it. We do not think that producing pseudo code to be interpreted is a satisfactory solution, because interpreting code is a very slow process. Our most important goal is to have the performance representations run faster than the programs they represent and reproduce faithfully the profiles of these programs.

There is a whole area where the interaction between the back end and the user can be very fruitful and which is beyond the scope of what can be done in an automatic way. This area could be called the area of the "metafunctions" supplied by the user. Consider our Example 2.4.1, where we have a program that reads an array of numbers and finds the sum of the

positive entries in it. Figure 2.4.3 shows the flow chart of such a program. In Chapter 3 we saw how to obtain an algorithmic ppf which should run faster than the corresponding skeleton because of the savings due to not having to evaluate the outer predicate. The final profile representation is $C_1B_1C_2B_2$, where C_1 represents the number of times the T-branch of the iteration is traversed (i.e., how many positive numbers are there in the array presented to the program) and C_2 represents the number of times the F-branch is traversed. If the user has some knowledge about the probability distribution of the sign of the numbers that will be presented to the program, then it may be possible for the user to provide such information to the back end. Thus, the performance representation could be modified to one that looks like $f(i)B_1g(i)B_2$, which would then require only one input, say the length of the string presented to the program, to produce the profile.

This approach may be helpful in determining the four basic quantities one would like to know about a program's running time:

<maximum, minimum, average, standard deviation> .

As was discussed in Section 1.3, our approach enables us to obtain exact points in the distribution of uniprogramming execution time of a given program. This may provide us with estimates on the four quantities mentioned above. The exact determination of them requires mathematical reasoning about the algorithm. However, if one has a complex program, obtaining points of the distribution of uniprogramming execution time may be very costly. Thus, our procedure helps us obtain more empirical information at a smaller computational cost. Judicious choice of input variables may give us sufficient information for the cases which one expects to encounter in practice.

A third module that can easily be incorporated into the system is a specialized data base. This subsystem would be queried and updated by the back end. Its purpose would be to store useful information gathered with experience. The system would then become an "expert system" in the sense that, after some time in use, its data base would have knowledge which was not there before. One example of items that should be stored would be user supplied factorizations of polynomials (or rational functions), and user supplied closed forms.

7.6. Summary

Our discussion throughout this work has been centered around the topic of building "performance representations" for programs. In this chapter we have presented some aspects which we had not dealt with before. In particular, we have discussed actions on variables which are not linear, control variables of types which are not numeric, and how to address the problem of verifying the hypotheses that allow us achieve our goal more efficiently than by using the skeleton. We have also sketched an implementation of these ideas. The main components to be implemented were seen to be a front end able to extract the data flow information needed to identify all the control variables, and a back end able to apply the techniques developed in the previous chapters of this work. An algebraic manipulation system like MACSYMA was seen to be of great help for several of the tasks to be performed by the back end.

CHAPTER 8

Microanalysis of Parallel Programs

In this chapter we shall explore the applicability of our methods to parallel programs. By parallel programs we shall understand programs written in programming languages which have explicit syntactic constructs permitting the coexistence of several sequential processes devoted to a common set of tasks. There are no *a priori* restrictions as to how these sequential processes are to be carried out, but (normally) they are in execution simultaneously.

Four examples of programming languages with this kind of constructs are PL/I with its multitasking facility, Nicklaus Wirth's MODULA programming language, Per Brinch Hansen's Concurrent Pascal programming language [Han77] and Narayana *et al.*'s CCNPASCAL [Nar79]. This type of parallelism in programs is sometimes called *explicit* parallelism.

In a program we may also have *implicit* parallelism, when an optimizing compiler produces code for a normal (sequential or single-thread of execution) program which will then have parts executable in parallel. The motivation for concurrent execution is speed. Of course, such *parallelised* program would execute faster in a multiprocessor system than in a single processor system. Our techniques do not deal with this type of performance optimization, even though there has been a substantial amount of interest and work in this area. One paper where a modeling technique for this type of parallelism is discussed is [Tow78].

The execution environment of a parallel program need not be that of a multiprocessor system. In fact, a parallel program may be executed in a single processor system managed according to the principles of multiprogramming: different parts of the program may be concurrently executing by sharing the processor and other resources of the system, sometimes even on a time-sliced basis. Indeed, we shall see that some of the difficulties encountered in analyzing parallel programs arise from the nature of the different execution environments.

8.1. The Loss of Sequentiality

We have characterized a parallel program as one consisting of sequential processes that are carried out simultaneously. The processes cooperate on common tasks by exchanging information through appropriate interprocess communication mechanisms. In the case of Concurrent Pascal, for example, they do so by exchanging data through shared variables [Han77]. Appropriate restrictions need to be imposed on the communication mechanisms to insure the consistency of the shared data at all times.

One problem with the loss of global sequentiality is that unrestricted access to the shared variables can make the result of a parallel program dependent on the relative speeds of its sequential processes. As this has the very negative side effect that it is possible to execute a parallel program several times using the same input data and obtain different results each time, interprocess synchronization mechanisms must be used.

When dealing with sequential programs, we need information about the processing environment only to establish relationships between the profile of a given run and performance indices which are compiler/system dependent. In particular, obtaining the profile of a run never requires information about

the processing environment of the analyzed program. This may not be the case for a parallel program. If the synchronization of the different sequential processes which make up a parallel program is dependent on the actual processing environment, then, since deriving the profile requires representing the synchronization mechanism, knowledge about the processing environment of the analyzed program will be needed. This fact makes analyzing the performance of parallel programs a harder task than that for sequential programs. We are now faced with the additional problem of reproducing the synchronization mechanism existing among the sequential processes.

This increase in difficulty coincides with the experience of all other fields of Computer Science which deal with concurrent processes: that the complexity of their analysis is much greater than that for sequential processes. To mention just one example, in the area of program correctness the work by Gries and Owicki [Gri77, Owi76a, Owi76b] shows the difficulties encountered when dealing with techniques for proving parallel programs correct.

In a parallel program all computations performed by any given sequential process between statements which require information or acknowledgement of other processes' actions can be analyzed by our earlier techniques. Thus, if each sequential process of a parallel program is executing on a dedicated processor, to find the profile we are reduced to the problem of reproducing the exchange of information through the synchronizing mechanisms in the system. This requires knowledge of each processing environment and of the interprocess communication links.

If several sequential processes are executing in a single processor, the overall system activity generated by them is dependent on the scheduling algorithm of the processor. Appropriate synchronization mechanisms leave the profile of a run not affected by this multiprogramming environment, but deriving those performance indices which depend on the activity of the whole system (like execution time) requires this information. Hence, to derive some performance indices of a parallel program one now needs to consider processes clustered by processors and describe their activity taking into account each processor's scheduler.

We see that the level of difficulty increases substantially in the latter case. A satisfactory modeling effort requires complete information about the processors capabilities as well as about the interprocessor communications. Accuracy is crucial because of events which are dependent on the relative completion times of other events. Our techniques are useful in determining the timing of events within one processor. The overall modeling, however, requires techniques which are very different in nature from the ones discussed in earlier chapters.

8.2. Some Models of Parallel Computations

In the previous section we have seen the necessity of modeling the environment in which a parallel program is to execute. For this reason, we shall now present an overview of models and concepts used in describing systems in which concurrent activity of sequential processes takes place.

The central issue in the discussions found in the literature is that of communication and synchronization between processes working towards a common goal. The component processes must be able to communicate and synchronize with each other. Many methods of achieving this have been

proposed. One widely adopted method of communication is by inspection and updating of a common store. However, this can create severe problems in the construction of correct programs and it may lead to expensive and unreliable implementations with some hardware technologies. A variety of methods have been proposed for synchronization: semaphores, events, conditional critical regions, monitors and queues, and path expressions are among the best known. Each one of these is demonstrably adequate for its purpose, but there is no widely accepted criterion for choosing among them. In [Hoa78], Hoare departs from the standard approach to the problems of interprocess communication and synchronization and proposes a way to deal with them based on the assumption that input, output and concurrency should be regarded as primitives of programming.

We now present some models of concurrent programming which have been proposed in recent years. We shall follow the treatment in [Mac79], where these models are presented in a rather formal and machine independent way. We first present three generic types of models and then three specific proposals.

8.2.1. Automata Models

These models generally consist of a state space, representing the possible states of the entire system, together with transmission functions which (nondeterministically) generate state sequences representing computations. They are designed for the investigation of automata-theoretic questions such as the decidability of certain formal properties of systems. To simplify proofs, it is desirable to idealize the model by minimizing and simplifying its structure. Unfortunately, the result is a "low level" model in which the "high level" phenomena of interest to a programmer cannot be directly and realistically represented.

Another drawback of typical automata models is that parallelism is reduced to the set of all possible interleavings of computation steps, where each step is represented by its incremental effect on the global state. This makes it difficult to subdivide a computation into the separate activities of independent processes.

6.2.2. Petri Nets

One of the most successful and intensively studied models of parallelism is the Petri net. This is a graph model which, as usually interpreted, represents elementary events in a computation and the way they depend on one another. Petri nets have been used to analyze parallelism in a wide variety of contexts, from hardware to operating systems, and have been applied to problems in the social sciences. They have also been used to describe the semantics of path expressions, which, as we mentioned earlier, are a tool for interprocess synchronization.

The appeal of Petri net models is based in a number of factors: they are simple and elegant in structure, and their graphical nature provides a visual aid to intuition. They are a good tool to study the fundamental nature of parallelism. They deal directly with questions concerning causality and dependence between events. They are capable of modeling phenomena at many levels of abstraction, from hardware to high-level languages. Relevant phenomena such as deadlock can be modeled fairly clearly and naturally.

However, Petri net models have some limitations, mostly due to their being idealized "low level" models. Expressing some problems (such as readers and writers synchronization) requires features which are not available in Petri nets. Nets are global representations of a system, and there are

no simple or obvious means of decomposing a net into subnets which represent natural subsystems. Conversely, there is a lack of natural composition operators for building complex nets out of simpler nets. Petri nets have a static structure, which makes it difficult to model the changing structure of a dynamic system. One can attempt to capture dynamic structures by the use of infinite nets, but the result is liable to be a rather obscure representation. These nets are suitable for describing the control aspects of computations, but provide no direct way of describing the flow of data or data-dependent conditional behavior.

6.2.3. Operating Systems Theory Models

This category is meant to include theoretical formulations of the problems of operating systems together with assorted techniques or language features, such as semaphores, critical regions, monitors, and path expressions, which have been proposed to deal with typical problems in systems programming. Most of these ideas are relevant to distributed computing but there are some differences in outlook.

In conventional computer systems, it is normally assumed that the task of the operating system is to prevent undesirable interactions between unrelated programs which are required for economic reasons to share the resources of the machine. The emphasis is on managing contention and preventing interference between processes. When dealing with a parallel program, on the other hand, contention over shared resources is less of a problem and we are more concerned with facilitating co-operation and communication between processes. In view of these differences, we should expect concurrent computing to require new concepts beyond those derived from experience with multiprogramming operating systems.

We now present three specific ways to view concurrent processing environments within the operating systems model category.

8.2.3.1. The Actor Model

The actor model is a behavioral approach to computation. *Actors* are self-contained program units which communicate by sending and receiving *messages*. The receipt of a message is called an *event*. Its effect is to activate the target actor, which then performs some internal computation leading to the sending of further messages and perhaps the creation of new actors. The aim of the actor model is to analyze the behavior of actor systems in terms of the causal and "incidental" relations between the events of a computation. In this respect it resembles the theory of causal nets studied by Petri, but is more specific and concrete.

The actor model, which has been developed over the past few years by Hewitt and his students at MIT, was inspired in part by the class notion of SIMULA 67. This notion combines the passive structural and the active procedural aspects of a data object in a single unit. Research on modeling programming language features using the lambda calculus was another contributing influence, as was the technique of continuation passing developed for applications in language semantics [Str74, Rey77].

A complete description of the model is given in [Mac79] or [Hew77] and will not be included here. One can hierarchically descr be the actor model by introducing first *basic actors*, which are those which cannot be decomposed into systems of simpler actors, and then build all actors and messages from them. Even messages can be considered special kinds of actors. In this model, no mention is made on the way communication takes place.

8.2.3.2. Communicating Behaviors

In [Mil77, Mil78a, Mil78b], Milner has developed an algebraic theory for describing and synthesizing systems of communication agents. Intuitively, the model deals with systems of computing agents communicating via input ports which are connected by channels. The state of an agent is represented by an abstract object called *behavior*, which expresses the *potential* communication activities of the agent. Each act of communication causes a change in the behaviors of the agents involved. Communication takes the form of a value-passing act requiring simultaneous co-operation between a sender and a receiver. Hence we can assume that the communication channels have no storage and are unidirectional (though in effect synchronization information is exchanged in both directions when communication takes place).

The mathematical theory of Milner is concerned only with the structure and semantics of behaviors. Formal behaviors may be interpreted as mathematical objects in a number of ways, including *processes*, which are the behavioral counterparts of mathematical functions, and *synchronization trees*, which represent behaviors in which communication is reduced to pure synchronization.

Each agent is assumed to possess a fixed, finite set of input and output ports. The input ports are labeled by *names* and the output ports are labeled by *conames*. The names and conames together constitute the set of labels, Λ. Labels are a device for specifying the interconnection of ports, and therefore of agents (and behaviors). Pairs of ports with complementary labels are assumed to be connected by a channel, and can therefore communicate.

The objective of the communicating behaviors model is to describe an algebra of behaviors, in which complex behaviors can be built up from simpler ones by the use of behavioral operators. One starts by considering constructions for expressing the dynamic aspects of a behavior. They include the primitives for communication and for expressing nondeterministic alternatives. One calls behaviors defined by means of these constructions "elementary" to distinguish them from those behaviors constructed in terms of them. Roughly speaking, the elementary behaviors are the states of single agents, while compound behaviors represent the compound states of networks of agents. A complete presentation of the model can be found in [Mil78b].

It is interesting to observe that in this model communication links (or channels) are an integral part of the description of the model. This forces synchronization restrictions which were not present in the actor model. These conditionants must be taken into account when reproducing the behavior of a parallel program in this environment.

6.2.3.3. Process Networks

The last model to be outlined involves networks of processes communicating by means of dedicated channels with storage. Because processes are isolated from time-dependent information, their semantics can be expressed "denotationally" in terms of functional relationships between entire input and output histories, represented by *streams*. Networks are constructed by functional composition and recursion, so they inherit the deterministic, functional nature of processes.

Processes are self-contained, independent modules, each executing its own sequential program accessing its own local store, which may be of

unbounded size. Processes only communicate with one another via channels, using input and output operations provided for the purpose. Channels transmit values (which may be restricted to a given type) from a unique producer process to a consumer process. They are assumed to provide unbounded buffering and to preserve the order of transmission.

An input operation may involve waiting for the producer, but this is made transparent to the consumer, so that time-dependent phenomena cannot affect the outcome of the computation. For the same reason, there is no "polling" operation to determine availability of input in a channel. This decision to hide all necessary synchronization within the input/output primitives has two important effects:

(1) Computation is determinate, in the sense that the sequence of output values depends only on the sequence of input values.

(2) A variety of scheduling strategies may be used without materially altering the outcome of the computation.

A beneficial effect of avoiding nondeterminacy is that processes have a straightforward functional nature. The communication history of each output channel (i.e., the sequence of values transmitted) is a function of the communication histories of the input channels. The model then is totally described by dealing with histories and with how to express functions on histories. Histories are represented by data of type "stream".

Nondeterminacy can be incorporated into the model by providing a polling primitive *next?* with which one determines whether the next input value in a stream is immediately available. Nondeterminacy needs to be incorporated into the model if we are to represent real-time applications. The full model can be found in [Kel77].

8.3. A Particular Case: Concurrent Pascal

Concurrent Pascal is an example of a programming language designed for real-time applications [Han77]. Concurrent Pascal extends Sequential Pascal with *concurrent processes*, *monitors* and *classes*. *system* types added to Sequential Pascal to represent these new entities are: type process, type monitor and type class.

A process type defines a data structure and a sequential statement that can operate on it. A monitor type defines a data structure and the operations that can be performed on it by concurrent processes. These operations can synchronize processes and exchange data among them. A class type defines a data structure and the operations that can be performed on it by a single process or monitor. These operations provide controlled access to the data.

To ease synchronization between processes, the type *queue* has been introduced. It may be used within a monitor type to delay and resume the execution of a calling process within a routine entry. At any time no more than one process can wait in a single queue. A variable of type queue can only be declared as a *permanent variable* within a monitor type.

The compiler prevents some time-dependent programming errors by checking that the private variables of one process are inaccessible to another. Processes can only communicate by means of monitors. A monitor defines all the possible operations on a shared data structure. It can, for example, define the send and receive operation on a message buffer. The compiler will check that processes only perform these two operations on a buffer. A monitor can delay processes to make their interactions independent of their speeds. A process that tries to receive a message from an

empty buffer will, for example, be delayed until another process sends a message to the buffer.

The structured way in which concurrent programming must be done in this language allows us to deal with the microanalysis of Concurrent Pascal programs in a three step procedure:

(1) For each sequential process within a parallel program a performance representation is built. The profile equations for each sequential process are thus expressed as a function of the input and shared variables. This allows us to analyze in isolation the behavior of any of the sequential processes which make up the parallel program.

(2) For each monitor we obtain its performance representation, where we have to make special provisions for the treatment of variables of type queue. The actions of the monitor on variables of type queue can not be predicted solely on the basis of syntactic information. Knowledge about the execution environment of the monitor (e.g., how are queues handled, types of delays that may occur when processing a message, arbitration mechanism for ties) and on the activities of several processes may be required to describe the effect of the monitor's action. This is the first time we encounter the phenomenon that some performance indices associated with the basic operations of a program are dependent on the results (or activities) of other processes.

(3) The system as a whole is simulated as a network where some nodes, those corresponding to monitors with variables of type queue, will have queues associated with them. Nodes with no queues associated with them will correspond to sequential processes. The arcs between nodes represent data paths. These paths may vary tremendously in nature:

they may be anything from an I/O channel to a telephone line. This is another instance where we are faced with an explicit dependence on the executing environment of the parallel program.

Thus, to reproduce the behavior of a parallel program, we are faced with the task of modeling its execution environment as well as representing each of its component parts. When we want to obtain performance indices such as the execution time, we need to know, as a function of the inputs, the time it will take for a sequential process to execute, the relevant data that it will produce, the time it will take that data to travel to a monitor which requires it, the time the monitor will take to process it, and the overall interrelationships existing between the different activities currently being processed in the system. In short, a full simulation of the network needs to be done. In this simulation, one may use the performance representations of each node to estimate the time spent at that node.

It should be clear that there will be situations where the modeling of the network will be easier than in others. One such case is when processors are never shared by distinct processes. The simplification arises because one does not have to deal with reproducing the scheduling decisions which need to be made when more than one process is running in one processor.

Unfortunately, at the time we write these lines, we have not had sufficient experience with these methods to report on any actual experiment. This whole area deserves further study, and we plan to report on it in the future. We are not aware of any other similar efforts in the prediction of program performance indices within distributed processing environments.

CHAPTER 9

Conclusions and Further Research

In this thesis we have studied the problem of finding efficient ways to determine, given the values for the input variables, the values of different performance indices associated with a program. We have seen that, for most purposes, we may concentrate on reproducing efficiently the dynamic profile of the program, i.e., on obtaining the exact profile for any run of the program as a function of the values of the input variables.

To achieve this, we have described several kinds of "performance representations" which express the profile equations of the program we are analyzing. We have shown that it is often possible to represent the profile equations by "program performance formulae" whose evaluation time is linear in the length of their expression. This is easily seen to be the best one can hope to obtain. We have also delimited the cases in which this can be done, and proposed some alternative methods for the other cases.

In fact, our "skeleton" procedure can always be used, in the case of sequential programs, to represent the profile equations of a program. It was noticed that, for compute bound sequential programs, the running time of the skeleton could be substantially shorter than that of running the actual program. Nevertheless, we also presented examples which showed that the running time of the skeleton need not be linear in the length of its text and that it could also be very close to the running time of the actual program.

A variety of solutions, and theoretical results which can guide us in their usage, were presented to overcome the different problems presented with the possible slowness of the skeleton, on the one hand, and the non-applicability of the program performance formulae, on the other hand. It was recognized that most of the "definability" problems are caused by the iterations. In particular, alternations within iterations often lead to the loss of optimal performance representation. We also examined in full detail the case when the actions on the control variables of an iteration were linear functions. In this case we saw that at run time we were even able to determine the exact pattern of truth values that a predicate would have.

We have not implemented a system which can build these performance representations for us. However, we have discussed what is needed to do so, and discovered that known techniques used in data flow analysis suffice to obtain all the information we need about the variables in a program. We believe that performance representations could be built by an "intelligent programming environment" at the same time that the program is being edited. This is one area of future research which deserves investigation.

We have also discovered that our methods can be used to obtain traces of programs efficiently. In fact, with minor modifications (which amount to write out where one is at each step as well as subscript values), the skeleton approach can be used to generate traces of programs, including data traces. Moreover, our techniques which yield faster performance representations can also be utilized to generate "condensed" traces, which can then be interpreted (or decoded) by using a simple postprocessor. The idea is that, when we are able to discover the behavior of an iteration as a function of the variables which control its looping, we may use this information to shorten the length of the trace without any loss of information. This method to obtain traces deserves further research, since no precedent approaches

seem to have been proposed. Its appeal is that, once the "trace representation" of a program is built, obtaining several traces should be much more economical than actually running an appropriately instrumented version of the program several times. Of course, the same post-processor could be used for traces obtained from any program by this method.

Another area which deserves further study is that of microanalysis of parallel programs. As we have seen, the problem of determining performance indices is much more complex in this case, and requires modeling of the computational environment of the parallel program. There has been little work done in this area, and our techniques do not prove to be directly applicable. They are useful for determining values of indices for a single process in a single processor, but we have not developed techniques to deal efficiently with the problem of representing the computational environment of a parallel program.

BIBLIOGRAPHY

[Aho74] Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[Aho79] Aho, Alfred V. and Ullman, Jeffrey D., *Principles of Compiler Design*, Addison Wesley (April 1979).

[All76] Allen, F.E. and Cocke, J., "A Program Data Flow Analysis Procedure," *Communications of the ACM* 19(3) pp. 137-147 (Mar. 1976).

[Art71] Artin, Emil, *Galois Theory*, University of Notre Dame Press (Jan. 1971).

[Ash71] Ashcroft, Edward and Manna, Zohar, *Translation of GO-TO Programs to WHILE Programs*, IFIP Congres, Ljubljana (Aug. 1971).

[Böh66] Böhm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with only two Formation Rules," *Communications of the ACM* 9(5) pp. 366-371 (May 1966).

[Boo57] Boole, George, *Finite Differences*, Chelsea Publishing Company, New York (1957). Fourth Edition

[Boo80] Booth, Taylor L. and Wiecek, Cheryl A., "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," *IEEE Transactions on Software Engineering* SE-6(2) pp. 138-151 (March. 1980).

[Che76] Cheatham, Thomas E. and Townley, Judy A., "Symbolic Evaluation of Programs, A look at Loop Analysis," pp. 90-98 in *Proceedings ACM Symposium on Symbolic and Algebra Computation*, (1976).

[Che78] Cheatham, Thomas E. and Washington, D., "Program Loop Analysis by Solving First Order Recurrence Relations," TR-13-78, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts (1978).

[Che79] Cheatham, Thomas E., Holloway, Glenn H., and Townley, Judy A., "Symbolic Evaluation and the Analysis of Programs," *IEEE Transactions on Software Engineering* SE-5(4) pp. 402-417 (July 1979).

[Coh74] Cohen, Jacques and Zuckerman, Carl, "Two Languages for Estimating Program Efficiency," *Communications of the ACM* 17(6) pp. 301-308 (June 1974).

[Coh76a] Cohen, Jacques and Roth, Martin, "On the Implementation of Strassen's Fast Multiplication Algorithm," *Acta Informatica* 6 pp. 341-355 (1976).

[Coh76b] Cohen, Ellis, "Program Reference for SPICE2," ERL-M592, ERL Memorandum, University of California, Berkeley (June 1976).

[Coh77a] Cohen, Jacques and Carpenter, Neal, "A Language for Inquiring about the Run-time Behaviour of Programs," *Software - Practice and Experience* 7 pp. 445-460 (1977).

[Coh77b] Cohen, Jacques and Katcoff, John, "Symbolic Solution of Finite-Difference Equations," *ACM Transactions on Mathematical Software* 3(3) pp. 261-271 (Sep. 1977).

[Coh79] Cohen, Jacques, Sitver, Robin, and Auty, David, "Evaluating and Improving Recursive Descent Parsers," *IEEE Transactions on Software Engineering* SE-5(5) pp. 472-480 (Sep. 1979).

[Coh65] Cohn, M. Richard, *Difference Algebra*, Interscience Publishers, New York (1965).

[Dav80] Davies, Anthony, "The Analogy Between Electrical Networks and Flowcharts," *IEEE Transactions on Software Engineering* SE-6(4) pp. 391-394 (July 1980).

[Dij68] Dijkstra, Edsger W., "Co-operating Sequential Processes," pp. 43-112 in *Programming Languages*, ed. Genuys, F., Academic Press, New York (1968).

[Dij72] Dijkstra, Edsger W., "Notes on Structured Programming," pp. 1-82 in *Structured Programming*, Academic Press, New York (1972).

[Dij75] Dijkstra, Edsger W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM* 18(8) pp. 453-457 (Aug. 1975).

[Dij78] Dijkstra, Edsger W., Lamport, Leslie, Martin, A. J., Scholten, C. S., and Steffens, E. F. M., "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM* 21(11) pp. 966-975 (Nov. 1978).

[End72] Enderton, H. B., *A Mathematical Introduction to Logic*, Academic Press (1972).

[Fat79] Fateman, Richard J., *Addendum to the Mathlab/MIT MACSYMA Reference Manual for VAX/UNIX VAXIMA*, Computer Science Division (December 1979).

[Fav79] Favaro, John M., "An Interactive Symbolic Executor Based on Macsyma," ERL 79/65, University of California, Berkeley, California (Dec. 1979). M.S. Research Project

[Fer78] Ferrari, Domenico, *Computer Systems Performance Evaluation*, Prentice-Hall (1978).

[Gos78] Gosper, R. W., "Decision Procedure for Indefinite Hypergeometric Summation," *Proceedings of the National Academy of Sciences, USA* 75(1) pp. 40-42 (Jan. 1978).

[Gri77] Gries, David, "An Exercise in Proving Parallel Programs Correct," *Communications of the ACM* 20(12) pp. 921-930 (Dec. 1977).

[Han77] Hansen, Per Brinch, *The Architecture of Concurrent Programs*, Prentice-Hall (1977).

[Han78] Hansen, Per Brinch, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM* 21(11) pp. 934-941 (Nov. 1978).

[Hen80] Hennell, Michael A. and Prudom, J. Alan, "A Static Analysis of the NAG Library," *IEEE Transactions on Software Engineering* SE-6(4) pp. 329-333 (July 1980).

[Hew77] Hewitt, C., "Viewing Control Structures as Patters of Passing Messages," *Artificial Intelligence* 8(3) pp. 323-364 (June 1977).

[Hoa78] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* 21(8) pp. 666-677 (Aug. 1978).

[How78] Howden, William E. , "DISSECT- A Symbolic Evaluation and Program Testing System," *IEEE Transactions on Software Engineering* SE-4(1) pp. 70-73 (Jan. 1978).

[Ing78] Ingalls, D. H. H., "The SMALLTALK-76 Programming System: Design and Implementation," pp. 9-15 in *Fifth Annual ACM Symposium on Principles of Programming Languages*, , Tucson (Jan. 1978).

[Ivi78] Ivie, John, "Some MACSYMA Programs for Solving Recurrence Relations," *ACM Transactions on Mathematical Software* 4(1) pp. 24-33 (Mar. 1978).

[Jol61] Jolley, L. B., , *Summation of Series*, Dover Publications Inc., New York (1961). Second Revised Edition

[Jor50] Jordan, Charles, *Calculus of Finite Differences*, Chelsea Publishing Company, New York (1950). Second Edition

[Kar79] Karr, Michael , *Summation in Finite Terms*, Massachusetts Computer Associates (Apr. 1979).

[Kel78] Keller, R. M., "Denotational Models for Parallel Programs with Indeterminate Operators," in *Formal Descriptions of Programming Languages*, ed. Gilchrist, B., North-Holland, Amsterdam (1978).

[Kin76] King, James C., "Symbolic Execution and Program Testing," *Communications of the ACM* 19(7) pp. 385-394 (July 1976).

[Knu71a] Knuth, Donald E., *Mathematical Analysis of Algorithms*, IFIP Congres, Ljublijana (Aug. 1971).

[Knu71b] Knuth, Donald E., "An Empirical Study of FORTRAN Programs," *Software - Practice and Experience* 1(1) pp. 105-133 (1971).

[Knu73] Knuth, Donald E. and Stevenson, F. R., "Optimal Measurement Points for Program Frequency Counts," *BIT* 13 pp. 313-322 (1973).

[Knu74] Knuth, Donald E., "Structured Programming with GOTO Statements," *Computing Surveys*, (6) pp. 261-301 (Dec. 1974).

[Knu78] Knuth, Donald E. and Jonassen, Arne T., "A Trivial Algorithm whose Analysis isn't," *Journal of Computer 'and Systems Sciences* 16(3) pp. 301-322 (1978).

[Kod78a] Kodres, Uno R., "Analysis of Real-Time Systems by Data Flowgraphs," *IEEE Transactions on Software Engineering* SE-4(3) pp. 169-178 (May 1978).

[Kod78b] Kodres, Uno R., "Discrete Systems and Flowcharts," *IEEE Transactions on Software Engineering* SE-4(6) pp. 521-525 (Nov. 1978).

[Lam78] Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM* 21(7) pp. 558-565 (July 1978).

[Led75] Ledgard, Henry F. and Marcotty, Michael, "A Genealogy of Control Structures," *Communications of the ACM* 18(11) pp. 629-639 (Nov. 1975).

[Lev61] Levy, H. and Lessman, F., *Finite Difference Equations*, McMillan, New York (1961).

[Mac79] MacQueen, David B., "Models for Distributed Computing," Rapport de Recherche, INRIA Laboria, Paris (Apr. 1979).

[Man74] Manna, Zohar, *Mathematical Theory of Computation*, McGraw-Hill Book Company (1974).

[Man78] Manna, Zohar and Waldinger, Richard, "The Logic of Computer Programming," *IEEE Transactions on Software Engineering* SE-4(3) pp. 199-229 (May 1978).

[Mat77] Mathlab, *MACSYMA Reference Manual*, Laboratory of Computer Science (December 1977).

[Mil77] Milner, R., "Flowgraphs and Flow Algebras," CSR-5-7, Department of Computer Science, University of Edinburgh, Edinburgh, England (July 1977).

[Mil78] Milner, R., "Algebras for Communicating Systems," CSR-25-78, Department of Computer Science, University of Edinburgh, Edinburgh, England (June 1978).

[Mil79] Milner, R., "Synthesis of Communicating Behavior," in *7th Symposium on Mathematical Foundations of Computer Science*, , Zakopane, Poland (1979).

[Mis78] Misra, Jayadev, "Some Aspects of the Verification of Loop Computations," *IEEE Transactions on Software Engineering* SE-4(6) pp. 478-488 (Nov. 1978).

[Moe77] Moenck, Robert, *On Computing Closed Forms for Summations*, 1977 MACSYMA User's Conference (1977).

[Nag75] Nagel, Lawrence W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," ERL-M520, ERL Memorandum, University of California, Berkeley (May 1975).

[Nar79] Narayana, K. T., Prasad, V. R., and Joseph, M., "Some Aspects of Concurrent Programming in CCNPASCAL," *Software - Practice and Experience* 9 pp. 749-770 (1979).

[Owi76a] Owicki, Susan and Gries, David, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica* 6 pp. 319-340 (1976).

[Owi76b] Owicki, Susan and Gries, David, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM* 19(5) pp. 279-285 (May 1976).

[Pra79] Pratt, Terrence, "Program Analysis and Decomposition Through Kernel-Control Decomposition," *Acta Informatica*, (9) pp. 195-218 (1979).

[Ram80] Ramamoorthy, C. V. and Ho, Gary S., "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering* SE-6(5) pp. 440-449 (Sep. 1980).

[Ram79] Ramshaw, Lyle H., *Formalizing the Analysis of Algorithms*, Stanford University (June 1979). Ph.D. Dissertation

[Rey65] Reynolds, J. C., *COGENT.ANL-7022*, Argonne National Laboratory, Argonne, Illinois (1965).

[Rey72] Reynolds, J. C., "Definitional Interpreters for Higher-order Programming Languages," in *Proceedings of the ACM Annual Conference*, (1972).

[Smi79] Smith, Connie U. and Browne, J. C., *Modeling Software Systems for Performance Predictions*, CMG X, Boulder (1979).

[Smi80] Smith, Connie U., "The Prediction and Evaluation of the Performance of Software From Extended Design Specifications," TR-154, University of Texas, Austin, Texas (Aug. 1980). Ph.D. Dissertation

[Str74] Strachey, C. and Wadsworth, C. P., "Continuations: A Mathematical Semantics for Handling Full Jumps," PRG-11, Programming Research Group, Oxford University, Oxford, England (Jan. 1974).

[Tow78] Towsley, D., Chandy, K. M., and Browne, J. C., "Models for Parallel Processing Within Programs: Application to CPU:I/O and I/O:I/O Overlap," *Communications of the ACM* 21(10) pp. 821-831 (Oct. 1978).

[Wat79] Waters, Richard C., "A Method for Analyzing Loop Programs," *IEEE Transactions on Software Engineering* SE-5(3) pp. 237-247 (May 1979).

[Weg75] Wegbreit, Ben, "Mechanical Program Analysis," *Communications of the ACM* 18(9) pp. 528-539 (Sep. 1975).

[Woo79] Woodward, Martin R., Hennell, Michael A., and Hedley, David, "A Measure of Control Flow Complexity in Progr :m Text," *IEEE Transactions on Software Engineering* SE-5(1) pp. 45-50 (Jan. 1979).

Appendix A

In this appendix we include the complete FORTRAN source of the file tmp.f and the subroutine MATLOC from SPICE.

The text of Appendix A is not included. It can be obtained directlly from the author.