

Copyright © 1981, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

FIRST ORDER LOGIC SYNTAX AND THE
DYNAMIC BEHAVIOR OF PROGRAMS

by

Luis Felipe Cabrera

Memorandum No. UCB/ERL M81/48

24 June 1981

ELECTRONICS RESEARCH LABORATORY
College of Engineering
University of California, Berkeley
94720

First Order Logic Syntax and the Dynamic Behavior of Programs

by

Luis Felipe Cabrera †

Department of Mathematics
and
Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

Abstract

A general purpose computer program can be viewed as a directed graph with five types of nodes and one type of arcs. Program semantics can be formalized by the *operational approach* which associates a specific action (given by a function) with each of the nodes. To study the *behavior* of a program as a function of its inputs, traditional methods require the use of the given program. This can be very inefficient. A new method is presented whereby the dynamic profile of a program is found efficiently. The notion of Skolem functions is embedded in our solution.

Given a program, a syntactic *performance representation* is built for it. Using this representation, we obtain the desired program behavior information much faster than by actually running the original program. For our solution, we exhibit the class of programs for which the behavior is determined in optimal time. This class is dependent on the semantics of our performance representations.

Our method may determine a maximal set of programs for which behavior information can be found in optimal time. We discuss this question in greater detail.

† The work reported here has been supported in part by the Computer Systems Design Program of the National Science Foundation under grants MCS-7824618 and MCS-8012900.

1. Introduction

The efficiency of computer programs is becoming a central issue in the implementation and utilization of novel ideas and techniques in various fields of Computer Science. Mathematical models of programs (and of their semantics) allow the study of programs in a programming-language independent way. In this paper we will study a syntax-oriented method which enables us to efficiently obtain performance information about a program. An open question is whether the class of programs for which we can find its behavior information in optimal time is maximal.

1.1. Motivation

It is known that the performance of a computer system depends on all of the aspects of its hardware and software configuration, as well as on the work load it has. It is thus advantageous to have programs which use appropriate algorithms and, most important, that make suitable usage of the resources available in their execution environment. Unfortunately, today there are no program design tools, or methodologies, which allow us to analyze a symbiosis of this kind between a program and its execution environment. It is thus of importance to have means of efficiently studying the behavior of programs.

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 implementations of a

given algorithm would become easier and less resource and time consuming.

1.2. Describing the Behavior of a Program Efficiently

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. However, 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 *dynamic 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 dynamic program profile. 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.

For example, if we are interested in counting the different kinds of atomic operations that the program performs, then we need the information 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 it to obtain the counts of the operations executed. This procedure is certainly

installation independent, thus, once this information has been found for a program, it never needs to be recomputed.

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.

This paper explores 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.

2. 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 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. In [Knu78] we can see that the complete analysis of a rather simple algorithm may require complex mathematical knowledge and expertise. The required amount of sophistication and level of reasoning about the program seems to go beyond the current level of what can be automated.

With a different approach, since 1974 Jacques Cohen and his collaborators have been *microanalyzing* 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.

The simple structure of many algorithms has proved that the method can yield interesting results. In [Coh78a] we see an analysis of Strassen'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.

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 the (fixed) probability that 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 intricate. 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.

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 no further information 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, 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 obtain evaluations which do not represent the execution of the program under any given set of inputs.

In [Weg75], the system Metric is presented. With it a very limited class of Lisp programs can be correctly microanalyzed. The highlights of Metric 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, and not the performance, of the analyzed programs.

3. Representation of Programs

We shall 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.

Definition 3.1

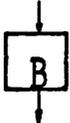
- (i) If  represents a basic block of statements in a program, then  is an elementary D-chart.



Figure 3.1

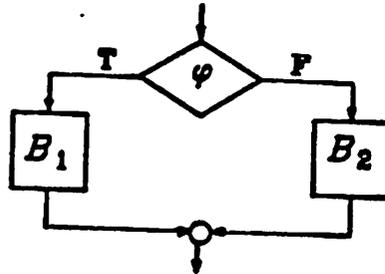


Figure 3.2

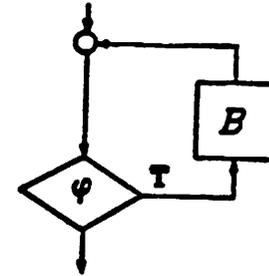


Figure 3.3

(ii) (Composition) If B_1 and B_2 are elementary D-charts, then Figure 3.1 is an elementary D-chart.

(iii) (Alternation) If B_1 and B_2 are elementary D-charts and φ a quantifier free formula in the language of arithmetic then Figure 3.2 is an elementary D-chart. We call the two branches the T-branch and the F-branch respectively. For example in Figure 3.2 we have the left branch as the T-branch and the right branch as the F-branch.

(iv) (Iteration) If B is an elementary chart and φ a quantifier free formula in the language of arithmetic, then Figure 3.3 is an elementary D-chart. We call the two branches the T-branch and the F-branch respectively. In Figure 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 3.2

A D-chart is a graph of the form $\begin{array}{c} \nabla \\ | \\ \boxed{B} \\ | \\ \nabla \end{array}$, where $\begin{array}{c} \downarrow \\ \boxed{B} \\ \downarrow \end{array}$ is an elementary D-chart.

Given an elementary D-chart  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 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 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.

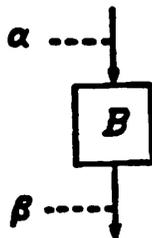


Figure 3.4

4. The Operational Semantics for Programs

To avoid syntax dependent definitions, one can view basic blocks of instructions as (black box) functions acting on the variables which appear in it. In this way, one associates with each rectangular node representing a basic block of instructions a function from variables to values in the domain of the variables. These functions are sometimes called *content* functions.

The extension of content functions to functions which describe the value of variables in any D-chart should be clear. It is done by induction on the complexity of the D-charts. We shall only present a sketch of this construction. First, it is assumed that all variables have a given value at the exit of the triangular node representing the entry point.

Say B is a basic block and γ the content function associated with it. Let $x[\alpha]$ denote the value of the variable x at the entry point α of B , and $x[\beta]$ its value at the exit point β of B . Then, $x[\beta] = \gamma(x[\alpha])$.

In the case of a branching with content function γ_1 for the T-branch, content function γ_2 for the F-branch and predicate φ , the value of x is modified by γ_1 if the branching predicate φ evaluates to true and by γ_2 otherwise.

Using this approach one makes precise the notion of values of variables at any given point in a D-graph, as well as the notion of exit or final value of a variable. We shall not discuss this formalization any further, but it should be clear that the notions of input variables, run, control variables and halting can be formalized within this framework.

5. Program Performance Formulae

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): $x_0, x_1, \dots, x_n, \dots$
- [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):

$$B_0, B_1, \dots, B_n, \dots$$

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.

5.1. 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:

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

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 5.1.1

The set of *terms* is the set of expressions generated from the constant symbols and variables by the operations Γ_{\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 5.1.2

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 $t_0 < t_1$.

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:

$$\Gamma_{-}(\varepsilon) = (-\varepsilon),$$

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

Definition 5.1.3

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

Definition 5.1.4

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, \dots 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: \leq , $>$, \geq . The longest definition in terms of $=$ and $<$ is for $>$:

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

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: $\&$, \rightarrow and \leftrightarrow . 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

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

will be written

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

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

- 1 for each variable x , $\mathbf{i}(x) = i(x)$.
- 2 $\mathbf{i}(0) = 0$ and $\mathbf{i}(1) = 1$.
- 3 If t_1, \dots, t_n are terms and f is one of \log , mod , $+$, $*$, then

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

where $f^{\mathbb{R}}$ is the operation defined in the real numbers which is denoted by f .

In particular, \times denotes the multiplication operation, i.e., $*^{\mathbb{R}}$ is \times .

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

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

where $f^{\mathbb{R}}: (\mathbb{R} \cup \{\infty\})^n \rightarrow \mathbb{R} \cup \{\infty\}$ 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 i , $\varphi[i]$ is the result of assigning values, via i , to all (free) variables in φ .

With atomic formulae,

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

With sentential formulae φ ,

$(\neg \varphi)[i]$ is true iff it is not the case that $\varphi[i]$ is true. $(\varphi_1 \text{ OR } \varphi_2)[i]$ is true iff $\varphi_1[i]$ is true or $\varphi_2[i]$ 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 5.2.1

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

1 for any special denotation symbol B_i , $I(B_i)[i] = 1B_i$.

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

$$I(\psi_1\psi_2)[i] = I(\psi_1)[i]I(\psi_2)[i].$$

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

$$\begin{aligned} I(\text{IF } \varphi . f(t_1, \dots, t_n) \text{ THEN } \psi_1 \text{ ELSE } \psi_2 \text{ FI})[i] & \text{ is equal to} \\ f^R(i(t_1), \dots, i(t_n)) \times I(\psi_1)[i] & \text{ if } \varphi[i] \text{ is true and equal to} \\ f^R(i(t_1), \dots, i(t_n)) \times I(\psi_2)[i] & \text{ if } \varphi[i] \text{ is false.} \end{aligned}$$

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 $f^R(i(t_1), \dots, i(t_n)) \times I(\psi_j)[i]$ is Λ . ■

Proposition 5.2.1

Let ψ be a program performance formula and i an assignment function of V into \mathbb{R} . Then $I(\psi)[i] = s_0s_1 \dots s_n$, 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. ■

5.3. 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 $\begin{array}{c} \downarrow \\ \boxed{B} \\ \downarrow \end{array}$, (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 $\begin{array}{c} \downarrow \\ \boxed{B_1} \\ \downarrow \end{array}$ and $\begin{array}{c} \downarrow \\ \boxed{B_2} \\ \downarrow \end{array}$ 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

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

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

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

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 α , 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 5.3.1

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

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 i will any $\#\varphi$ evaluate to ∞ and so ∞ will not appear in $I(\psi)[i]$. Conversely, if ∞ never appears for any assignment i , 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 i for its input variables \vec{x} , the sequence

$$a_0 B_0 a_1 B_1 \cdots a_n B_n$$

is the *profile* of P under input i iff, for $0 \leq j \leq n$, the run with inputs i 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 i , $I(\psi_D)[i]$ is the profile of P under input i . 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 5.3.2

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 i be an assignment function.

Clearly for an elementary D-chart of the form $\begin{array}{c} \downarrow \\ \boxed{B} \\ \downarrow \end{array}$ where $\begin{array}{c} \downarrow \\ \boxed{B} \\ \downarrow \end{array}$ is an indecomposable elementary D-chart, $\psi_D = \psi_P (= 1B_i$ where B_i is the special denotation symbol assigned to A). 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

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

and $I(\psi)[i]$ is $1 \times I(\psi_1)[i]$ if $\varphi[i]$ is true, and is $1 \times I(\psi_2)[i]$ if $\varphi[i]$ is false. Thus, in either case, we obtain that $\psi_D = \psi_P$ 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 5.3.3

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 $\psi_{D_{11}} = \psi_{P_{11}}$ and $\psi_{D_{12}} = \psi_{P_{12}}$. (see Figure 5.3.1). Then, $\psi_{D_1} = \psi_{P_1}$ iff the same branch of the alternation is traversed each time the T-branch of the iteration is traversed.

Proof

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

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

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

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

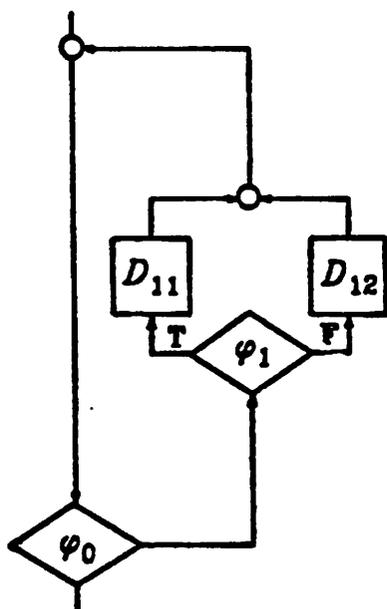


Figure 5.3.1

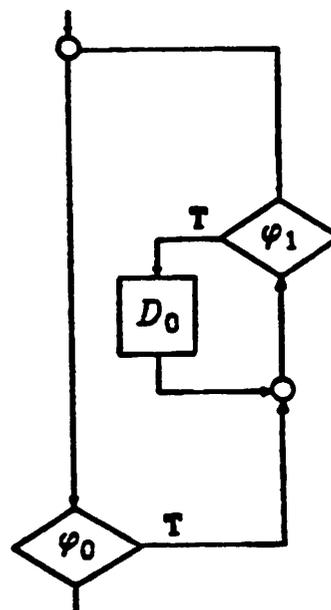


Figure 5.3.2

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

Lemma 5.3.4

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 5.3.2). We assume further that $\psi_{D_0} = \psi_{P_0}$. Then, $\psi_D = \psi_P$ 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 $\varphi_0[i]$ is true, the T-branch of the outer iteration is traversed $\#\varphi_0^R[i]$ times. If $\varphi_1[i]$ is true and we are traversing the outer loop's T-branch, then the inner one will be traversed $\#\varphi_1^R[i]$ times. Thus, $\#\varphi_0^R[i] \times \#\varphi_1^R[i] \times I(\psi_{D_{11}})[i]$ 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 $\#\varphi_0^R[i] \times \Lambda \times I(\psi_{D_{11}})[i]$, 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 $I(\psi)[i]$ are when both $\varphi_0[i]$ and $\varphi_1[i]$ are true, and when $\varphi_0[i]$ is true and $\varphi_1[i]$ is false. In the latter case $I(\psi)[i]$ 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 $\varphi_0[i]$ and $\varphi_1[i]$ are both true, then $I(\psi)[i]$ is $\#\varphi_0^R[i] \times \#\varphi_1^R[i] \times I(\psi_{D_0})[i]$, 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, the hypotheses of these lemmas have been implicitly made in all the literature we know about.

In a D-chart, whenever alternations within iterations satisfy the hypothesis of Lemma 5.3.3, we say that *alternations are well behaved*. Similarly if nested iterations satisfy the hypothesis of Lemma 5.3.4 we say that *iterations are well behaved*.

Theorem 5.3.5 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 5.3.3 or Lemma 5.3.4, 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 5.3.3 and 5.3.4, 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 5.3.1 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 , $\#\varphi_0^R[i] \times I(\psi_{D_1})[i]$ will appear if $\varphi_0[i]$ is true. But then, this is as in Lemma 5.3.3, 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 5.3.2, we argue that all iterations appearing in the D-chart must be well behaved. ■

Theorem 5.3.5 shows that our goal 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.

Example 5.3.1

In Figure 5.3.3 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 5.3.3 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.

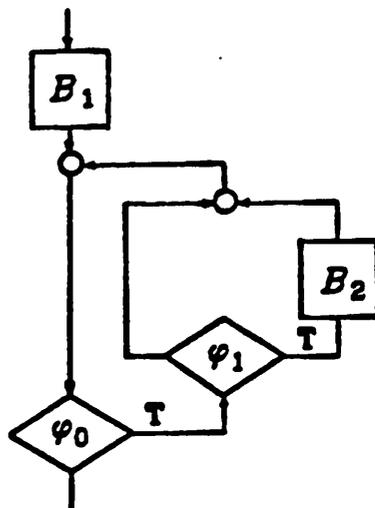


Figure 5.3.3

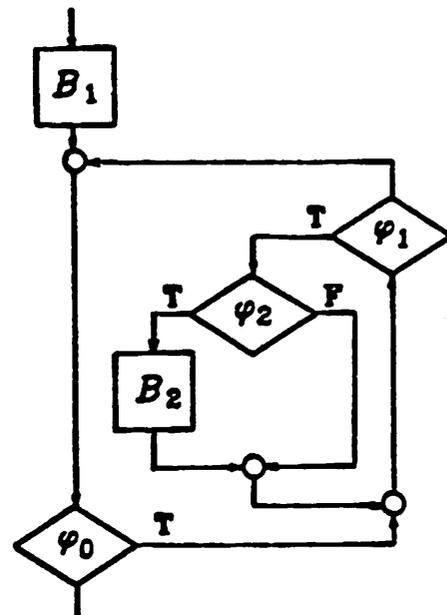


Figure 5.3.4

Example 5.3.2

Figure 5.3.4 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 j^{th} column up to the $A[1,j]^{\text{th}}$ one in the j^{th} entry of the array S . Thus, the outer predicate, φ_0 , is $i \leq M$, φ_1 is $j \leq A[1,i]$ and φ_2 is $A[i,j] \geq 0$. 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 k^{th} 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 \vec{x} ; thus $\vec{x}[\alpha]$ abbreviates $\langle x_1[\alpha], \dots, x_n[\alpha] \rangle$. k , when used as above, will be called the *iteration index*.

The hypothesis of Lemma 5.3.3 can also be characterized by a logical condition on the predicates φ_0 and φ_1 :

Theorem 5.3.6

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 5.3.1). Then, the same branch of the alternation is traversed each time the T-branch of the iteration is traversed iff for all assignment functions i , whenever the T-branch of the iteration is traversed, (1) $(\varphi_0(\vec{x}_0[\alpha]) \rightarrow \varphi_1(\vec{x}_1[\alpha]))$ true implies

$(\varphi_0(x_0[k]) \rightarrow \neg\varphi_1(x_1[k]))$ is false for all positive integers $k \leq \#\varphi_0^R(x_0[\alpha])$ or (2) $(\varphi_0(x_0[\alpha]) \rightarrow \neg\varphi_1(x_1[\alpha]))$ true implies $(\varphi_0(x_0[k]) \rightarrow \varphi_1(x_1[k]))$ is false for all positive integers $k \leq \#\varphi_0^R(x_0[\alpha])$.

Proof

Let i 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 $(\varphi_0(x_0[k]) \rightarrow \varphi_1(x_1[k]))$ would be true for all integers k , $0 \leq k \leq \#\varphi_0^R(x_0[\alpha])$. Thus, $(\varphi_0(x_0[k]) \rightarrow \neg\varphi_1(x_1[k]))$ would be false for all integers k , $0 \leq k \leq \#\varphi_0^R(x_0[\alpha])$.

For the "only if" part we argue as follows. Say (1) is true. Then $(\varphi_0(x_0[k]) \rightarrow \neg\varphi_1(x_1[k]))$ being false for all positive integers $1 \leq k \leq \#\varphi_0^R(x_0[\alpha])$ 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 i , 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 xR_0y , where R_0 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.

5.4. 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 5.4.1

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 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} , Γ_{\cdot} and Γ_{+} .

Special terms can be evaluated using the same interpretation function I introduced in Section 5.2.

Proposition 5.4.1

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

Proof

By induction on the complexity of special terms.

We notice, as in Theorem 5.3.1, 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 5.4.2

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

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

■

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 $\hat{\varphi}$ and $-\hat{\varphi}$. 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 n^{th} 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 5.4.1

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 $a^k x[0] + b \left(\frac{a^k - 1}{a - 1} \right)$.

Definition 5.4.3

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

Definition 5.4.4

For any variable x and any iteration L we say that $\tau(x, \gamma)$ is a *closed form* for x in L if $\tau(x, \gamma)$ 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, γ can be assumed to be of type integer. Example 5.4.1 depicts a closed form for x in the associated iteration L .

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

Definition 5.4.5

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 5.4.1, $\tau_{j,\beta}$ 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 5.4.2, $\tau_{j,\beta}$ is

$$\text{IF } (\varphi, 1) \text{ THEN } \tau_{j,\beta_1}\{\alpha_1/\alpha\} \text{ ELSE } \tau_{j,\beta_2}\{\alpha_2/\alpha\} \text{ FI}$$

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 5.4.3, $\tau_{j,\beta}$ is

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

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 5.4.2

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 .

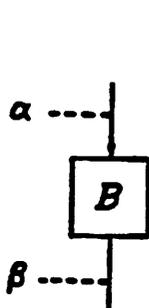


Figure 5.4.1

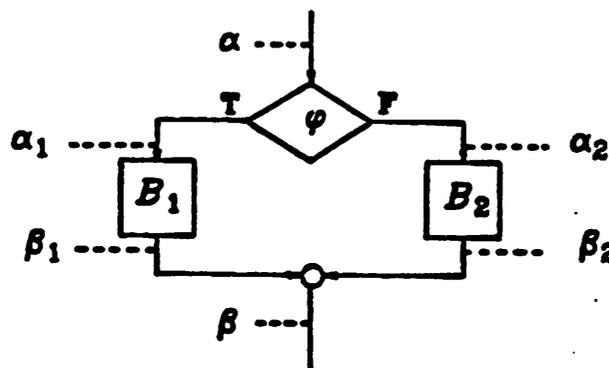


Figure 5.4.2

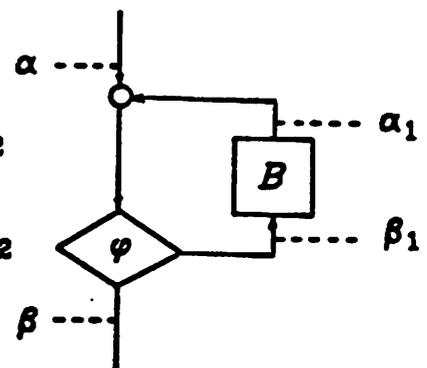


Figure 5.4.3

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]. 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.

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

Theorem 5.4.3 Definability Preservation for Variables

(i) If $\tau_{f,\alpha}$ is definable and D is an irreducible elementary D-chart, then $\tau_{f,\beta}$ 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 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 5.4.2, 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

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

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

(iii) For any assignment i , $I(\tau_{j,\beta})[i]$ is equal to $x_j[\beta]$ in the corresponding run, but this is equal to $x_j[\#\varphi^R[i]]$, where $x[0]$ is $I(\tau_{j,\alpha})[i]$. 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 5.4.3 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 5.4.4 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 5.4.5 Characterization of Definability for Variables

(i) For an irreducible D-chart D , $\tau_{j,\beta}$ is definable iff the hypothesis (i) of Theorem 5.4.3 or the hypothesis (i) of Theorem 5.4.4 hold.

(ii) For an alternation D , $\tau_{j,\beta}$ is definable iff the hypothesis (ii) of Theorem 5.4.3 or the hypothesis (ii) of Theorem 5.4.4 hold.

(iii) For an iteration D , $\tau_{j,\beta}$ is definable iff the hypothesis (iii) of Theorem 5.4.3 or the hypothesis (iii) of Theorem 5.4.4 hold.

Proof

Theorems 5.4.3 and 5.4.4 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 5.4.2 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 i , 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 i , 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 5.4.3 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 τ_1, \dots, τ_n 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 5.4.6

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

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 5.4.6

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 5.4.5 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 [Cab81], we see how we may deal with a wider family of programs by changing some of our syntactic constructs. The goal still is that of obtaining efficiently a representation of program profile equations, but the complexity of the evaluation will no longer be linear.

6. Conclusions

In this paper we have presented a new approach for the efficient generation of program profiles. The approach is to build performance representations of programs. From these representations one obtains all the desired performance information more efficiently than by using the program. A formal language in which performance representations of programs can be expressed was introduced, and the semantics for it were also given. We then studied conditions

under which optimally efficient performance representations could be obtained.

Theorem 5.3.5 characterizes the family of programs for which our method yields optimally efficient performance representations. It is seen that 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 5.3.1 and 5.3.2 show that in general we cannot expect to achieve much more than what our interpretation function I allows us to achieve.

It is an open question whether the family of programs determined by Theorem 5.3.5 is maximal if we consider all possible evaluation functions I which operate in a one-pass left-to-right manner. This question is also open if we only require I to have evaluation cost linear in the length of the expression to be evaluated.

Subsection 5.4 deals with the problem of finding definable programs. In this area it is not known for which recurrences one can have a decision procedure which yields a closed form. In [Cab81] one finds a discussion on this subject as well as some results on closed forms for conditional recurrence relations.

7. Acknowledgements

I thank Domenico Ferrari for introducing me to this area of Computer Science, to Steven Muchnick for showing me different formalizations of program semantics, and to the members of the PROGRES group for their useful comments and insights.

8. Bibliography

- [Boh66] Bohm, 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).
- [Cab81] Cabrera, Luis Felipe, "Syntax Oriented Analysis of the Run Time Performance of Programs," ERL Memorandum M81/30, University of California, Berkeley (May 13, 1981). Ph.D. Dissertation
- [Che76] Cheatham, Thomas E. and Townley, Judy A., "Symbolic Evaluation of Programs, A look at Loop Analysis," pp. 90-96 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).
- [Coh78] Cohen, Jacques and Roth, Martin, "On the Implementation of Strassen's Fast Multiplication Algorithm," *Acta Informatica* 6 pp. 341-355 (1978).
- [End72] Enderton, H. B., *A Mathematical Introduction to Logic*, Academic Press (1972).
- [Fer78] Ferrari, Domenico, *Computer Systems Performance Evaluation*, Prentice-Hall (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).
- [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 Congress, Ljubljana (Aug. 1971).
- [Knu71b] Knuth, Donald E., "An Empirical Study of FORTRAN Programs," *Software - Practice and Experience* 1(1) pp. 105-133 (1971).
- [Knu78] Knuth, Donald E. and Jonassen, Arne T., "A Trivial Algorithm whose Analysis isn't," *Journal of Computer and Systems Sciences* 18(3) pp. 301-322 (1978).
- [Weg75] Wegbreit, Ben, "Mechanical Program Analysis," *Communications of the ACM* 18(9) pp. 528-539 (Sep. 1975).