Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of$(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, $\cdots$, $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands

Language Protection by Trademark Ill-advised

Key Words and Phrases: TRAC languages, procedure-oriented language, proprietary software, protection of software, trademarks, copyright protection, patent protection, standardization, licensing, Mooers doctrine

CR Categories: 2.12, 2.2, 4.0, 4.2

EDITOR:

I would like to comment on a policy published 25 August 1967 by the Rockford Research Institute Inc., for trademark control of the TRAC language "originated by Calvin N. Mooers of that corporation": "It is the belief at Rockford Research that an aggressive course of action can and should be taken to protect the integrity of its carefully designed languages." Mr. Mooers believes that "well-drawn standards are not enough to prevent irresponsible deviations in computer languages," and that therefore "Rockford Research shall insist that all software and supporting services for its TRAC languages and related services be furnished for a price by Rockford, or by sources licensed and authorized by Rockford in a contract arrangement." Mooers' policy, which applies to academic institutions as well as commercial users, includes "authorized use of the algorithm and primitives of a specific TRAC language; authorization for experimentation with the language . . ."

I think that this attempt to protect a language and its software by controlling the name is very ill-advised. One is reminded of the COMIT language, whose developers (under V. Yngve) restricted its source-level distribution. As a result, that effort was bypassed by the people at Bell Laboratories who developed SNOBOL. This latter language and its software were inevitably superior, and were immediately available to everyone, including the right to make extensions. Later versions benefitted from "meritorious extensions" by "irrepressible young people" at universities, with the result that SNOBOL today is an important and prominent language, while COMIT enjoys relative obscurity.

Mr. Mooers will find that new TRAC-like languages will appear whose documentation, because of the trademark restriction, cannot mention TRAC. Textbook references will be similarly inhibited. It is unfortunate.

BERNARD A. GALLER
University of Michigan
Ann Arbor, Mich. 48104

Mr. Mooer's Reply

EDITOR:

Professor Galler's letter, commenting on our Rockford Research policy statement on software protection of 25 August 1967, opens the discussion of what may be a very significant development to our computing profession. This policy statement applies to our TRAC (TM) computer-controlling languages. The statement includes a new doctrine of software protection which may be generally applicable to a variety of different kinds of complex computer systems, computer services, languages, and software. Already it is evident that this doctrine has a number of interesting legal and commercial implications. It is accordingly appropriate that it be subject to critical discussion.

The doctrine is very simple. For specificity, I shall describe it in regard to the TRAC languages which we have developed: (1) Rockford Research has designated itself as the sole authority for the development and publication of authentic standards and specifications for our TRAC languages; and (2) we have adopted TRAC as our commercial trademark (and service mark) for use in connection with our computer-controlling languages, our publications providing standards for the languages and any other related goods or services.

The power of this doctrine derives from the unique manner in which it serves the interests of the consuming public—the people who will be using computer services. The visible and recognized TRAC trademark informs this public—the engineers, the sociology professors, the business systems people, and the nonprogrammers everywhere—that the language or computer capability identified by this trademark adheres authentically and exactly to a carefully drawn Rockford Research standard for one of our TRAC languages or some related service. This is in accord with a long commercial and legal tradition.

The evils of the present situation and the need to find a suitable remedy are well known. An adequate basis for proprietary software development and marketing is urgently needed, particularly in view of the doubtful capabilities of copyright, patent, or "trade secret" methods when applied to software. Developers of valuable systems—including languages—deserve to have some vehicle to give them a return. On the user side the nonexistence of standards in the computer systems area is a continuing nuisance. The proliferation of dialects on valuable languages (e.g. SNOBOL or FORTRAN) is sheer madness. The layman user (read "nonprogrammer") who now has access to any of several dozen computer facilities (each with incompatible systems and dialects) needs relief. It is my opinion that this new doctrine of autonomous standardization coupled with resort to commercial trademark can provide a substantial contribution to remedying a variety of our problems in this area.

Several points of Professor Galler's letter deserve specific comment. The full impact of our Rockford Research policy (and

indeed of this doctrine applied to other developments) upon academic activities cannot be set forth in just a few words (cf. Rockford Memo V-202). It is my firm belief that academic experimentation must be encouraged—indeed it cannot be stopped. Nevertheless, the aberrant or even possibly improved products coming from the academic halls must not be permitted to confuse or mislead the consuming public. Careful use of, and respect for, trademark can ensure that this does not occur.

SNOBOL was mentioned as illustrating a presumably desirable situation regarding innovation. Yet according to the *Snobol Bulletin No. 3* (November 1967), we find already a deep concern regarding serious incompatibilities among the many "home-made" implementations of this language. In addition, there are serious complaints over the profound lack of "upward compatibility" between the latest "SNOBOLS." The consequent inability of the users to exchange or publish useful algorithms is cited. These are exactly some of the problems that our policy hopes to avoid.

The future of our computing technology lies in service to the layman users. Our present chaos in interfaces, formats, lack of standards, proliferation of needless dialects, unreliable documentation, and all the other hazards and incompatibilities is completely intolerable to the users. The users know it. It is about time we knew it too.

I believe that this doctrine of autonomous standardization and trademark identification is a long step forward in service to the user public, and thus is in the right direction. According to the almost uniformly favorable response we have received to date, many others seem to think the same way. I expect to see the doctrine have wide application.

CALVIN N. MOOERS
Rockford Research Institute Inc.
Cambridge, Mass. 02138

No Trouble with Atlas I Page-Turning Mechanism

Key Words and Phrases: Atlas I, page-turning procedures
CR Categories: 4.2, 4.22

EDITOR:

The editorial on "The European Computer Gap," *Comm. ACM 10* (April 1967), 203, tells of "paper designs that could never be converted into operational systems," and among these includes "the page-turning procedures proposed with the original design of the Atlas."

Not merely does this do injustice to Atlas, but it is in fact quite wrong. The Atlas I machine we have here has a one-level store made up of 48K words of 2μs cores and 96K words on drums. The paging and page-turning mechanism have worked without any trouble at all almost from the beginning—so well that it is something we hardly ever think about. To give an idea of how intensively the system is used let me say that since 1964 we have been running a service for research workers in all British universities with a very mixed load of programs in all the major languages. We put about 2500 jobs through the machine each week, and the system efficiency is around 70 percent. By this last figure I mean that, of all the instructions obeyed by the machine over a long period, 70 percent goes into either the compiling or execution of users' programs. The figure can rise to over 90 percent with a favorable job-mix.

J. HOWLETT
Atlas Computer Laboratory
Chilton, Didcot
Berkshire, England

On Practicality of Sieving Techniques vs. the Sieving Algorithm

Key Words and Phrases: prime numbers, sieving algorithms, sieving techniques, indexing techniques
CR Categories: 3.15, 5.39

EDITOR:

After reading the remarks on the sieving algorithms in the September 1967 issue of *Communications of the ACM* [p. 569], I should like to point out the fact that these algorithms are presented in ALGOL solely for the purpose of communicating the idea of the algorithm, and that the published running times for the sieving algorithms are not representative of the sieving process.

For practical use these algorithms are usually implemented in assembly language on machines with high speed index registers, since the sieving technique is essentially an indexing technique. For example, an algorithm which, when given an array of length n, sieves between p and $p+2n$ was implemented in the assembly language for an IBM 360 model 40. This algorithm assumes only that the even numbers between p and $p+2n$ have already been crossed out; it does not incorporate any of the special features of Algorithms 310 and 311. The time required to compute the first m primes is given in the following table.

m	*Time* (sec)
10,000	7
100,000	87
500,000	525
1,000,000	1149
1,250,000	1487

The value of n used in preparing the above table was $n = 16{,}000$. The average time for sieving over an interval of length 32,000 was 2.46sec.

Thus, while it may appear that the sieving algorithms are too slow to be practical when implemented in a compiler language, the above times indicate that the sieving technique can be practical when implemented in an assembly language.

JOHN E. HOWLAND
University of Oklahoma
Norman, Oklahoma 73069

Dealing with Neely's Algorithms

Key Words and Phrases: algorithm, computation of statistics, truncation error, Neely's comparisons
CR Categories: 4.0, 5.5, 5.11

EDITOR:

When we decided to use the method of Welford [1] in our FORTRAN programs we made some comparisons, but arrived at a conclusion which contradicts Peter Neely's [2]. This was an invitation to us to scrutinize Neely's work. His remark, "The inaccuracy noted for M_2 may be due to IBM-FORTRAN, which does not compile a floating round," is one pointer to the source of inaccuracy. Indeed, with a compiler which does compile a floating round, Welford's method gives results equivalent to those obtained with the two-pass method recommended by Neely. If a floating round is not compiled, the use of Kahan's trick [3] will give excellent results even on those machines, such as an IBM 1620 which truncates before normalizing a floating point sum.

Another source of inaccuracy, however, is due to the way Welford's formulas are programmed. In particular we found that the formulas as given by Welford and programmed by Neely are not the best available.

The best versions for programming purposes seem to be the following:

$$m_0 = 0;\quad m_i = m_{i-1} + (x_i - m_{i-1})/i,\quad i = 1, n;\quad M_2 = m_n \qquad (1)$$

$$s_0 = 0;\quad s_i = s_{i-1} + (x_i - m_{i-1})^2 - (x_i - m_{i-1})^2/i,\quad i = 1, n;\quad S_3 = s_n \qquad (2)$$

and P_3 similar to (2). Of these equations (1) is most important and addition using Kahan's trick will give an error-free answer.

Not using Kahan's trick will give results for variable $x_{i,10}$ not as good as those obtained with the two-pass method, but since we think this kind of variable is not likely to occur in practical work, we recommend (1) and (2) for calculation of the mean and corrected sum of squares. Since we found that from (1) and (2) Σx and Σx^2 are more accurately retrieved than when computed directly, we think that (1) can be used in numerical integration too, if the result afterwards is multiplied by the number of intervals.

REFERENCES:

1. WELFORD, B. P. Note on a method for calculating corrected sums of squares and products. *Technometrics IV* (1962), 419–420.
2. NEELY, PETER M. Comparison of several algorithms for computation of means, standard deviations and correlation coefficients. *Comm. ACM 9*, 7 (July 1966), 496–499.
3. KAHAN, W. Further remark on reducing truncation errors. *Comm. ACM 8*, 1 (Jan. 1965), 40.

A. J. VAN REEKEN
Rekencentrum
Katholieke Hogeschool
Tilburg, The Netherlands

Abbreviations for Computer and Memory Sizes

Key Words and Phrases: memory, thousand
CR Categories: 2.44, 6.34

EDITOR:

The fact that 2^{10} and 10^3 are almost but not quite equal creates a lot of trivial confusion in the computing world and around its periphery. One hears, for example, of doubling the size of a 32K memory and getting a 65K (not 64K) memory. Doubling again yields a 131K (not 130K) memory. People who use powers of two all the time know that these are approximations to a number they could compute exactly if they wanted to, but they seldom take the trouble. In conversations with outsiders, much time is wasted explaining that we really can do simple arithmetic and we didn't mean exactly what we said.

The confusion arises because we use K, which traditionally means 1000, as an approximation for 1024. If we had a handy name for 1024, we wouldn't have to approximate. I suggest that κ (kappa) be used for this purpose. Thus a 32κ memory means one with exactly 32,768 words. Doubling it produces a 64κ memory which is to say one with exactly 65,536 words. As memories get larger and go into the millions of words, one can speak of a $32\kappa^2$ (33,554,432-word) memory and doubling it will yield a $64\kappa^2$ (67,108,864-word) memory. Users of the language will need to have at their fingertips only the first nine powers of 2 and will not need to explain the discrepancies between what they said and what they meant.

DONALD R. MORRISON
Computer Science, Division 5256
Sandia Corporation, Sandia Base
Albuquerque, N. Mex.

In Defense of Langdon's Algorithm*

Key Words and Phrases: lexicographic permutation
CR Categories: 5.39

EDITOR:

Ord-Smith [Letter to the Editor, *Comm. ACM 10*, 11 (Nov. 1967), 684] makes some impertinent remarks on the subject of Langdon's algorithm [1]. The main point of the letter "that there does not appear to be any combinatorial advantage of circular ordering over lexicographic ordering" is hardly relevant. The problem attacked by Langdon is not to find *combinatorial* advantage but rather *computational* advantage, which Langdon's algorithm most certainly provides.

Most of the score or so of ALGOL algorithms published in CACM on the subject of lexicographic succession have been badly written; they contain only the sketchiest of theoretical discussion, and the obscurity of their construction masks their essentially simple methodology. In contrast, Langdon gives a clear and concise theoretical discussion and logic diagram. The relative brilliance of Langdon's paper may be taken as an indication that formal papers and logic diagrams are a superior method for presentation of this subtle type of arithmetic. The essential point that Ord-Smith seems to have missed is that Langdon's algorithm uses rotation rather than transposition as the basis of iteration, thus taking advantage of the hardware design of modern computers which perform rotation much more efficiently than transposition. The ALGOL language, however, does not give the user access to the rotation registers and hence will not implement this algorithm efficiently with respect to running time. The fact that the transposition methods give shorter running times indicates not superior algorithms but a fundamental weakness of the ALGOL language for this type of numeric manipulation. Given access to the rotation registers, Langdon's algorithm would be efficient in both coding compactness and running time.

REFERENCE:

1. LANGDON, G. J. An algorithm for generating permutations. *Comm. ACM 10*, 5 (May 1967), 298–299.

B. E. RODDEN
Defence Research Establishment
Toronto, Ontario, Canada

*DRET Technical Note No. 686

Endorsing the Illinois Post Mortem Dump

Key Words and Phrases: ALCOR post mortem dump
CR Categories: 4.12, 4.42

EDITOR:

The authors of "The ALCOR Illinois 7090/7094 Post Mortem Dump" [*Comm. ACM 10*, 12 (Dec. 1967), 804–808] have presented a technique for producing post mortem dumps which, in my opinion, should be incorporated in all high level programming languages. A similar technique has been in operation for several years at Manchester [1] and has proved to be extremely useful, especially for student programmers.

REFERENCE:

1. BROOKER, R. A., ROHL, J. S., AND CLARK, S. R. The main features of Atlas Autocode. *Comput. J. 8* (Jan. 1966), 303–310.

S. R. CLARK
Department of Computing Science
The University of Manitoba
Winnipeg, Canada

Computational Linguistics

D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

CORRADO BÖHM AND GIUSEPPE JACOPINI
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines λ and R. That family is a proper subfamily of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now still enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermes [3], Ciampa [4], Riguet [5], Ianov [6], Asser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

This paper was presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory, Jerusalem, Israel. Preparation of the manuscript was supported by National Science Foundation Grant GP-2880.

This work was carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC) in collaboration with the International Computation Centre (ICC), under the Italian Consiglio Nazionale delle Ricerche (CNR) Research Group No. 22 for 1963–64.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object x of a set X, the former of which may be imagined concretely as the set of the digits contained in the memory of a computer, the tape configuration of a Turing machine, etc. There are other boxes of predicative type (see Figure 2) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of $x \in X$ occurs. Examples of diagrams are: $\Sigma(\alpha, \beta, \gamma, a, b, c)$ [Figure 3] and $\Omega_5(\alpha, \beta, \gamma, \delta, \epsilon, a, b, c, d, e)$ [see Figure 4]. It is easy to see a difference between them. Inside the diagram Σ, some parts which may be considered as a diagram can be isolated in such a way that if $\Pi(a, b)$, $\Omega(\alpha, a)$, $\Delta(\alpha, a, b)$ denote, respectively, the diagrams of Figures 5–7, it is natural to write

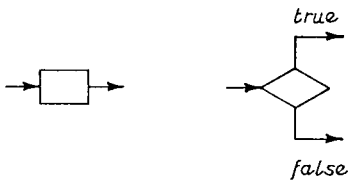
$$\Sigma(\alpha, \beta, \gamma, a, b, c) = \Omega(\alpha, \Delta(\beta, \Omega(\gamma, a), \Pi(b, c))).$$

Nothing of this kind can be done for what concerns Ω_5; the same happens for the entire infinite class of similar diagrams

$$\Omega_1[=\Omega], \Omega_2, \Omega_3, \cdots, \Omega_n, \cdots,$$

whose formation rule can be easily imagined.

Let us say that while Σ is decomposable according to subdiagrams Π, Ω and Δ, the diagrams of the type Ω_n are not decomposable. From the last consideration, which should be obvious to anyone who tries to isolate with a



FIGS. 1–2. Functional and predicative boxes

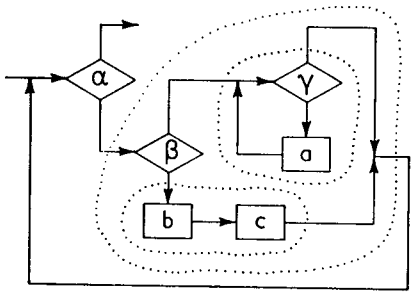


FIG. 3. Diagram of Σ

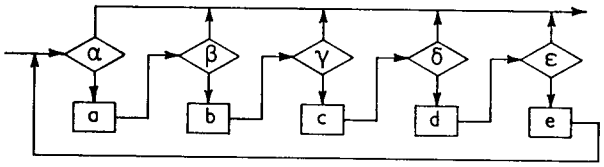
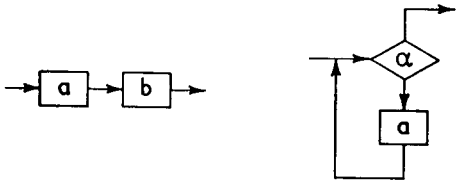
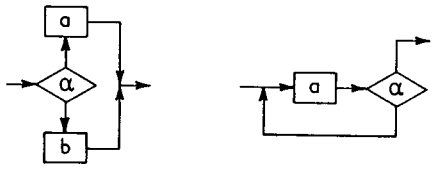


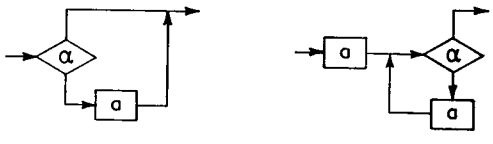
FIG. 4. Diagram of Ω_5



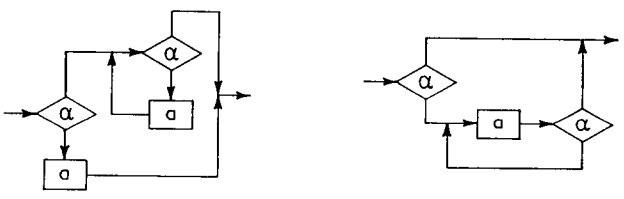
FIGS. 5–6. Diagrams of Π and Ω



FIGS. 7–8. Diagrams of Δ and Φ



FIGS. 9–10. Diagram of Λ and a diagram equivalent to Φ



FIGS. 11–12. Diagrams equivalent to Λ and Ω

broken line (as was done for Σ) a part of Ω_n provided with only one input and one output, it follows that:

It is not possible to decompose all flow diagrams into a finite number of given base diagrams.

However, together with this decomposition, that could be called *strong*, another decomposition may be considered which is obtained by operating on a diagram equivalent to the one to be decomposed (that is, the diagram has to express the same transformation, whatever the meaning of the boxes contained in it may be). For instance, it may be observed that if we introduce $\Phi(\alpha, a)$ [as in Figure 8] and $\Lambda(\alpha, a)$ [as in Figure 9] the diagrams of Φ, Λ and Ω become, respectively, equivalent to Figures 10, 11 and 12.

Thus, the following decompositions may be accepted:

$$\Phi(\alpha, a) = \Pi(a, \Omega(\alpha, a))$$

$$\Lambda(\alpha, a) = \Delta(\alpha, \Omega(\alpha, a), a)$$

$$\Omega(\alpha, a) = \Lambda(\alpha, \Phi(\alpha, a)).$$

Nevertheless, it is to be reckoned that the above statement holds even with regard to the new wider concept of decomposability. In fact, it does not seem possible[1] for every Ω_n to find an equivalent diagram which does not contain, as a subprogram, another Ω_n or an Ω of higher order. For instance, note that

$$\Omega_3(\alpha, \beta, \gamma, a, b, c) = \Lambda(\alpha, \Pi(a, \Omega_3(\beta, \gamma, \alpha, b, c, a)))$$
$$= \Omega_6(\alpha, \beta, \gamma, \alpha, \beta, \gamma, a, b, c, a, b, c)$$

and similar formulas hold for all orders of Ω.

The proved unfeasibility is circumvented if a new predicate is added and if, among the elementary operations, some are assumed which either add one bit of information to the object of the computation or remove one from it. The extra bits have a stack structure (formally described below as nested ordered pairs) since it is sufficient to operate and/or take decisions only on the topmost bit.

Therefore, three new functional boxes denoted by T, F, K, and a new predicative box ω are introduced. The effect of the first two boxes is to transform the object x into the ordered pair (v, x) where v can have only the values **t** (true) or **f** (false); more precisely,

$$x \xrightarrow{T} (\mathbf{t}, x), \quad x \xrightarrow{F} (\mathbf{f}, x), \quad (\mathbf{t}, x) \xrightarrow{T} (\mathbf{t}, (\mathbf{t}, x))$$

and so on. Box K takes out from an ordered pair its second component

$$(v, x) \xrightarrow{K} x, \quad (\mathbf{t}, (\mathbf{f}, (\mathbf{t}, x))) \xrightarrow{K} (\mathbf{f}, (\mathbf{t}, x)).$$

The predicate ω is defined as

$$\omega[(v, x)] = \mathbf{t} \Leftrightarrow v = \mathbf{t},$$

i.e., the predicate ω is verified or not according to whether the first component of the pair is T or F; ω and K are defined only on a pair; on the contrary, all the boxes

[1] We did not, however, succeed in finding a plain and sufficiently rigorous proof of this.

$\alpha, \beta, \gamma, \cdots, a, b, c, \cdots$ operating on x are not defined on a pair. The following statement holds:

If a mapping $x \rightarrow x'$ is representable by any flow diagram containing $a, b, c, \cdots, \alpha, \beta, \gamma, \cdots$, it is also representable by a flow diagram decomposable into Π, Φ *and* Δ *and containing the same boxes which occurred in the initial diagrams, plus the boxes K, T, F and ω.*

That is to say, it is describable by a formula in $\Pi, \Phi, \Delta, a, b, c, \cdots, T, F, K, \alpha, \beta, \gamma, \cdots, \omega$.

NOTE. A binary switch is the most natural interpretation of the added bit v. It is to be observed, however, that in certain cases if the object x can be given the property of a list, any extension of the set X becomes superfluous. For example, suppose the object of the computation is any integer x. Operations T, F, K may be defined in a purely arithmetic way:

$$x \xrightarrow{T} 2x + 1, \quad x \xrightarrow{F} 2x, \quad x \xrightarrow{K} \left[\frac{x}{2}\right]$$

and the oddity predicate may be chosen for ω. The added or canceled bit v emerges only if x is thought of as written in the binary notation system and if the actions of T, F, K, respectively, are interpreted as appending a one or a zero to the far right or to erase the rightmost digit.

To prove this statement, observe that any flow diagram may be included in one of the three types: I (Figure 13), II (Figure 14), III (Figure 15), where, inside the section lines, one must imagine a part of the diagram, in whatever way built, that is called $\mathfrak{A}$ or $\mathfrak{B}$ (not a subdiagram). The branches marked 1 and 2 may not always both be present; nevertheless, from every section line at least one branch must start.

As for the diagrams of types I and II, if the diagrams in Figures 16–17, are called A and B,[2] respectively, I turns into Figure 20 and may be written

$$\Pi(\Pi(T, \Phi(\omega, \Pi(\Pi(K, a), A))), K)$$

and II turns into Figure 21, which may be written

$$\Pi(\Pi(T, \Phi(\omega, \Pi(K, \Delta(\alpha, A, B)))), K).$$

The case of the diagram of type III (Figure 15) may be dealt with as case II by substituting Figure 22, where $\mathfrak{C}'$ indicates that subpart of $\mathfrak{C}$ accessible from the upper entrance, and $\mathfrak{C}''$ that part accessible from the lower entrance.

If it is assumed that A and B are, by inductive hypothesis,[3] representable in Π, Φ and Δ, then the statement is demonstrated.

It is thus proved possible to completely describe a program by means of a formula containing the names of diagrams Φ, Π and Δ. It can also be observed that Ω, Π and Δ could be chosen, since the reader has seen (see

[2] If one of the branches 1 or 2 is missing, A will be simply Figure 18*a* or 18*b*, and similarly for B. If the diagram is of the type of Figure 19 where $V\{\in\}(T, F)$, it will be simply translated into $\Pi(V, A^*)$ where A^* is the whole subdiagram represented by $\mathfrak{A}$.

[3] The induction really operates on the number $3N + M$, where M is the number of boxes T and F in the diagram and N is the number of all boxes of any other kind (predicates included).

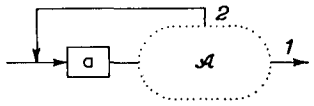


FIG. 13. Structure of a type I diagram

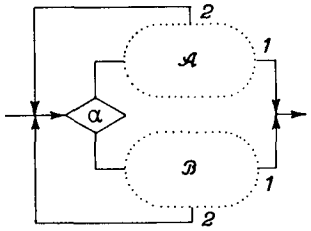


FIG. 14. Structure of a type II diagram

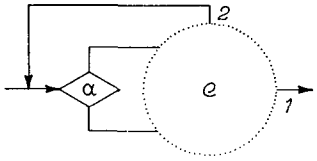


FIG. 15. Structure of a type III diagram

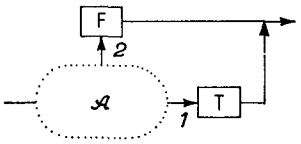


FIG. 16. A-diagram

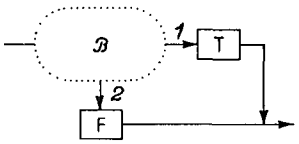


FIG. 17. B-diagram

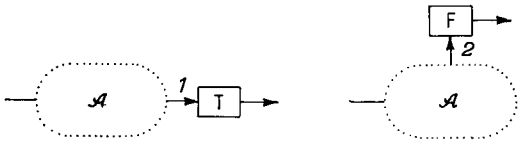


FIG. 18*a–b*. Two special cases of the A-diagram

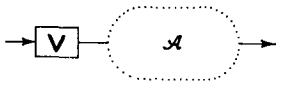


FIG. 19. Diagram reducible to $\Pi(V, A^*)$

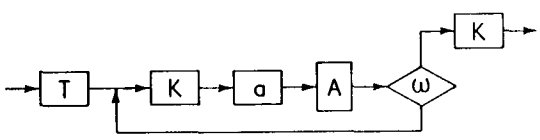


FIG. 20. Normalization of a type I diagram

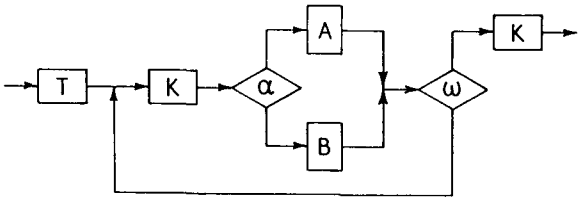


FIG. 21. Normalization of a type II diagram

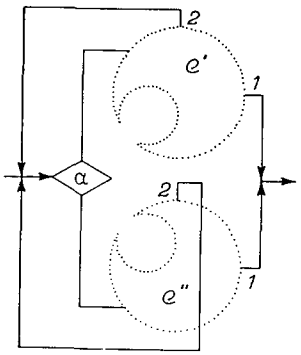


FIG. 22. Normalization of a type III diagram

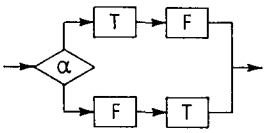


FIG. 23. The diagram $\underline{\alpha}$

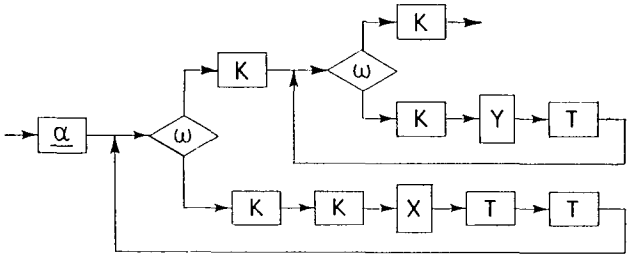


FIG. 24. Transformation of $\Delta(\alpha, X, Y)$

formula and Figure 10) that Φ can be expressed using Ω and Π. Moreover, it is observed that the predicate ω occurs only as the first argument of Φ (or, if desired, of Ω) and all the others as arguments of Δ: $\Phi(\omega, X)$ and $\Delta(\alpha, X, Y)$, etc.

Now let us define for every predicative box $\alpha, \beta, \cdots$ a new functional box $\underline{\alpha}, \underline{\beta}, \cdots$ with the following meaning:

$$\underline{\alpha} \equiv \Delta(\alpha, \Pi(T, F), \Pi(F, T)) \qquad \text{(Figure 23)}$$

This simplifies the language. In fact, any $\Delta(\alpha, X, Y)$ can be replaced by (see also Figure 24):

$$\Pi(\underline{\alpha}, \Pi(\Omega(\omega, \Pi(K, \Pi(K, \Pi(X, \Pi(T, T))))), \Pi(K, \Pi(\Omega(\omega, \Pi(K, \Pi(Y, T))), K)))).$$

Then we can simply write:

$$\Pi(X, Y) \equiv XY,$$

$$\Pi(\Pi(X, Y), Z) = \Pi(X, \Pi(Y, Z)) \equiv XYZ,$$

owing to the obvious associativity of Π. We may also write:[4]

$$\Omega(\omega, X) \equiv (X).$$

[4] The same notation is followed here as in [8].

To sum up: every flow diagram where the operations $a, b, c, \cdots$ and the predicates $\alpha, \beta, \gamma \cdots$ occur can be written by means of a string where symbols of operations $a, b, c, \cdots, \underline{\alpha}, \underline{\beta}, \underline{\gamma}, \cdots, T, F, K$ and parentheses (,) appear. For example:

$$\Pi(a, b) = ab$$

$$(^5)\quad \Omega(\alpha, a) = \underline{\alpha}K(Ka\underline{\alpha}K)K$$

$$(^5)\quad \Phi(\alpha, a) = F(Ka\underline{\alpha}K)K$$

$$\Delta(\alpha, a, b) = \underline{\alpha}(KKaTT)K(KbT)K$$

$$(^5)\quad \Lambda(\alpha, a) = \underline{\alpha}K(KaT)K$$

$$(^5)\quad \Omega_2(\alpha, \beta, a, b) = F(K\underline{\alpha}(KTT)K(Ka\underline{\beta}(KTT)K(KbFT))K)K.$$

More abstractly, the main result can be summarized as follows. Let

X be a set of objects x

Ψ a set of unary predicates $\alpha, \beta, \cdots$ defined in X

O a set of mappings $a, b, \cdots$ from X to X

$\mathfrak{D}(\Psi, \mathrm{O})$ the class of all mappings from X to X describable by means of flow diagrams containing boxes belonging to $\Psi \cup \mathrm{O}$.

Y the set of objects y defined by induction as follows:

$$\begin{cases} X \subset Y \\ y \in Y \Rightarrow (\mathbf{t}, y) \in Y, (\mathbf{f}, y) \in Y \end{cases} \qquad (1)$$

ω a predicate, defined in Y (at least on $Y - X$) by

$$\begin{cases} \omega(\mathbf{t}, x) = \mathbf{t} \\ \omega(\mathbf{f}, x) = \mathbf{f} \end{cases}$$

T, F two mappings defined on Y by

$$T[y] = (\mathbf{t}, y)$$

$$F[y] = (\mathbf{f}, y)$$

K a mapping defined in Y by

$$K[(\mathbf{t}, y)] = K[(\mathbf{f}, y)] = y$$

$\underline{\Psi}$ a set of mappings $\underline{\alpha}, \underline{\beta}, \cdots$ defined on X, with values in Y as follows:

$$\underline{\alpha}[x] = \neg\alpha[x], (\alpha[x], x) \qquad (2)$$

etc.

Now, given a set Z of objects z, a set Q of mappings from Z to Z, and *one* unary predicate π defined in Z, let us recursively define for every $q \in Q$ a new mapping $^{\pi}(q)$, written simply (q) if no misunderstanding occurs, as

[5] These formulas have not been obtained using the general method as described. The application of that method would make the formula even more cumbersome.

follows:

$$\begin{cases} \pi[z] \to (q)[z] = z \\ \sim\pi[z] \to (q)[z] = (q)[q[z]]. \end{cases}$$

For every $q_1, q_2 \in Q$, let us call q_1q_2 the mapping defined by $q_1q_2[z] = q_2[q_1[z]]$. Let us call $\mathcal{E}(\pi, Q)$ the class of mappings from Z to Z defined by induction as follows:

$$\begin{cases} Q \subset \mathcal{E}(\pi, Q) \\ q \in \mathcal{E}(\pi, Q) \Rightarrow (q) \in \mathcal{E}(\pi, Q) \\ q_1 \in \mathcal{E}(\pi, Q), q_2 \in \mathcal{E}(\pi, Q) \Rightarrow q_1q_2 \in \mathcal{E}(\pi, Q). \end{cases}$$

Note the following useful properties of $\mathcal{E}$:

$$Q_1 \subset Q_2 \to \mathcal{E}(\pi, Q_1) \subset \mathcal{E}(\pi, Q_2) \tag{3}$$

$$Q_2 \subset \mathcal{E}(\pi, O_1) \to \mathcal{E}(\pi, O_1 \cup Q_2) = \mathcal{E}(\pi, Q_1). \tag{4}$$

The meaning of the last statement can easily be rewritten:

$$\mathfrak{D}(\Psi, O) \subset \mathcal{E}(\omega, O \cup \Psi \cup \{T, F, K\}). \tag{5}$$

3. Applications to the Theory of Turing Machines

In a previous paper [8], a programming language $\mathcal{P}'$ was introduced which described, in a sense specified in that paper, the family $\mathfrak{B}'$ of Turing machines for a (leftward) infinite tape and any finite alphabet $\{c_1, c_2, \cdots, c_n\} \cup \{\square\}$, where $n \geqq 1$, $\square$ is the symbol for the blank square on the tape. Using the notation of Section 2 (see Note),

$$\mathfrak{B}' \equiv \mathfrak{D}(\{\alpha\}, \{\lambda, R\}) \tag{6}$$

where

- α is the unary predicate true iff the square actually scanned (by the Turing machine head) is blank (i.e. contains $\square$);
- λ is the operation of replacing the scanned symbol c_i with c_{i+1} ($c_0 \equiv c_{n+1} \equiv \square$) and shifting the head one square to the left;
- R is the operation of shifting the head one square, if any, to the right.

Briefly, α is a predicate, λ and R are partially defined functions[6] in the set X of tape configurations. By "tape configuration" of a Turing machine is meant the content of the tape plus the indication of the square being scanned by the machine head.

Example. If the configuration (at a certain time) is

$$x \equiv \cdots \square\square c_1\underline{c_n}c_n,$$

then

$$\alpha[x] \equiv \mathbf{f}, \quad \lambda[x] \equiv \cdots \square\square\underline{c_1}\square c_n, \quad R[x] \equiv \cdots \square\square c_1c_n\underline{c_n}$$

where the underscore indicates the scanned square. In [8] a language $\mathcal{P}''$ (describing a proper subfamily of Turing machines) has been shown. It was defined as follows.

(*i*) $\lambda, R \in \mathcal{P}''$ (Axiom of Atomic Operations)

(*ii*) $q_1, q_2 \in \mathcal{P}''$ implies $q_1q_2 \in \mathcal{P}''$ (Composition Rule)

(*iii*) $q \in \mathcal{P}''$ implies $(q) \in \mathcal{P}''$ (Iteration Rule)

(*iv*) Only the expressions that can be derived from (*i*), (*ii*) and (*iii*) belong to $\mathcal{P}''$.

Interpreting q_1, q_2 as functions from X to X, q_1q_2 can be interpreted as the composition $q_2 \circ q_1$, i.e.

$$q_1q_2[x] \equiv q_2[q_1[x]] \qquad x \in X$$

and (q) can be interpreted as the composition of q with itself n times: $q \circ \cdots \circ q \equiv q^n$, i.e. $q^n[x] \equiv q[\cdots[q[x]]\cdots]$ where $q^0[x] = x$ and $n = \mu\nu\{\alpha[q^\nu[x]] = \mathbf{t}\}$, $\nu \geqq 0$, i.e. (q) is the minimum power of q (if it exists) such that the scanned square, in the final configuration, is $\square$.

From the point of view of this paper, the set $\mathfrak{B}''$ of the configuration mappings described by $\mathcal{P}''$ is

$$\mathfrak{B}'' \equiv \mathcal{E}(\alpha, \{\lambda, R\}). \tag{7}$$

The drawbacks of $\mathcal{P}''$ as opposed to $\mathcal{P}'$ are that not all Turing machines may be *directly* described by means of $\mathcal{P}''$. For instance, it was proved in [8] that the operation H^{-1} (performed by the machine, which does nothing if the scanned symbol is different from $\square$, and otherwise goes to the right until the first $\square$ is scanned) cannot be described in $\mathcal{P}''$ ($H^{-1} \notin \mathfrak{B}''$).

Nevertheless, the most surprising property of $\mathcal{P}''$ is that, according to the commonest definition of "computing" a function by a Turing machine, every partial recursive function f in $m \geqq 0$ variables can be evaluated by a program $P_f \in \mathcal{P}''$ (see [8]).

Although this last property enables us to build a one–one mapping (via a gödelization of the Turing machines) of $\mathcal{P}'$ in $\mathcal{P}''$, it is here preferred to find a more direct correspondence between Turing machines, without any reference to partial recursive functions. To every Turing machine M, let us associate the machine M^* whose initial (and final) tape configuration is obtained by interspersing a blank square between every two contiguous squares of the tape of M. During the computation, these auxiliary squares are used to record, from right to left, the values v of the switch stack.

More precisely, for every configuration $x \equiv \cdots\square u_1 \cdots u_{K-1}\underline{u_K}u_{K+1}\cdots u_m$ where $u_i \in \{\square, c_1, \cdots, c_n\}$, let us call x^* the configuration

$$x^* \equiv \cdots \square\square\square u_1 \cdots \square u_{K-1}\square u_K\square\underline{u_{K+1}} \cdots \square u_m.$$

If M designates the Turing machine which when applied[7] to configuration b gives e as the final configuration, i.e. if $M[b] = e$, then M^* is a machine such that $M^*[b^*] = e^*$.

We want to prove: $M \in \mathfrak{B}', \Rightarrow M^* \in \mathfrak{B}''$.

Taking advantage of the theorem (5), we may write

$$\mathfrak{B}' \subset \mathcal{E}(\omega, \{\lambda, R, \underline{\alpha}, T, F, K\}). \tag{8}$$

Following the definition (1) of Y, the mapping $x \to x^*$

[6] For more details, see [8, 9].

[7] For simplicity, as in (6), Turing machines and configuration mappings will be identified.

is now extended to a mapping $y \to y^*$ as follows:

$$\text{if } y^* \equiv \cdots \square u_{K-1} \square \underline{u}_K \cdots \Rightarrow$$
$$(\mathbf{t},\ y)^* \equiv \cdots \square \underline{u}_{K-1} \square u_K \cdots, \quad (9)$$
$$(\mathbf{f},\ y)^* \equiv \cdots \square \underline{u}_{K-1} c_1 u_K \cdots.$$

Obviously,

$$M \in \mathfrak{B} \Rightarrow M \in \mathcal{E}(\omega, \{\lambda, R, \underline{\alpha}, T, F, K\})$$

and therefore

$$M^* \in \mathfrak{B}'^* \Rightarrow M^* \in \mathcal{E}(\omega^*, \{\lambda^*, R^*, \underline{\alpha}^*, T^*, F^*, K^*\}).$$

It is only necessary to prove that

$$\mathcal{E}(\omega^*, \{\lambda^*, R^*, \underline{\alpha}^*, T^*, F^*, K^*\}) \subset \mathcal{E}(\alpha, \{\lambda, R\}).$$

First, observe that for every machine $Z^* \in \mathcal{E}(\omega^*, \{\cdots\})$,

$$^{\omega^*}(Z^*) \equiv R\ ^{\alpha}(LZ^*R)L,$$

where $L \equiv [\lambda R]^n\lambda$ is the operation of shifting the head one square to the left, has been proved; therefore,

$$\mathcal{E}(\omega^*, \{\cdots\}) \subset \mathcal{E}(\alpha, \{\cdots\} \cup \{\lambda, R\}).$$

Secondly, it can be easily checked that

$$\{\lambda^*, R^*, T^*, F^*, K^*\} \subset \mathcal{E}(\alpha, \{\lambda, R\}).$$

In fact,

$$\lambda^* = \lambda L, \qquad R^* = R^2, \qquad T^* = L^2,$$
$$F^* = L\lambda, \qquad K^* = R(\lambda R)R.$$

According to (4), it has been proved that

$$\mathcal{E}(\omega^*, \{\cdots\}) \subset \mathcal{E}(\alpha, \{\underline{\alpha}^*, \lambda, R\}).$$

Thirdly,

$$\underline{\alpha}^* \in \mathcal{E}(\alpha, \{\lambda, R\}).$$

From formula (2) and the convention (9) it must follow that:

$$\underline{\alpha}^*[\cdots \square u_2 \square u_1 \square \underline{\square} \cdots] = \cdots \square \underline{u_2} c_1 u_1 \square \square \cdots \quad (10)$$

$$\underline{\alpha}^*[\cdots \square u_2 \square u_1 \square \underline{c} \cdots] = \cdots \square \underline{u_2} \square u_1 c_1 c \cdots \quad (11)$$

where $u_1, u_2 \in \{\square, c_1, \cdots, c_n\}$ and $c \in \{c_1 \cdots, c_n\}$.

In order to implement $\underline{\alpha}^*$, the program $L^3\lambda$ [which meets conditions (10)] must be merged with $L\lambda L^2$ [which meets (11)]. A solution[8] can be written in the form

$$\underline{\alpha}^* \equiv L^3\lambda R^4(X)L^5(Y)R,$$

which obviously satisfies (10). The program (X) can be chosen mainly to copy the symbol c on the first free blank square; the program (Y), to execute the inverse operation, i.e.,

$$(X)[\cdots \square u_2 c_1 u_1 \square \underline{c} \cdots] = \cdots \underline{c} u_2 \square u_1 c_1 \square \cdots$$
$$(Y)[\cdots\ c u_2 \square u_1 c_1 \square \cdots] = \cdots \square \underline{u_2} \square u_1 c_1 c \cdots$$

[8] The authors are indebted to the referee for this solution, which is shorter and more elegant than theirs.

It is not difficult to test that the choice

$$(X) \equiv (r'L(\lambda R)\lambda L(\lambda R)L^2\lambda R^6),$$
$$(Y) \equiv (r'R^5\lambda L^4),$$

where $r' \equiv [\lambda R]^n$, gives the desired solution.

RECEIVED NOVEMBER, 1965

REFERENCES

1. PETER, R. Graphschemata und rekursive Funktionen. *Dialectica 12* (1958), 373–393.
2. GORN, S. Specification languages for mechanical languages and their processors. *Comm. ACM. 4*, (Dec. 1961), 532–542.
3. HERMES, H. *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit.* Springer Verlag, Berlin, 1961.
4. CIAMPA, S. Un'applicazione della teoria dei grafi. *Atti del Convegno Nazionale di Logica, Torino 5–7* (April 1961), 73–80.
5. RIGUET, J. Programmation et théorie des catégories. Proc. ICC Symp. *Symbolic Languages in Data Processing*, Gordon and Breach, New York, 1962, 83–98.
6. IANOV, YU, I. On the equivalence and transformation of program schemes. *Dokl. Akad. Nauk SSSR 113*, (1957), 39–42. (Russian).
7. ASSER, G. Functional algorithms and graph schema. *Z. Math. Logik u. Grundlagen Math., 7*, (1961), 20–27.
8. BÖHM, C. On a family of Turing machines and the related programming language. *ICC Bull. 3*, (July 1964), 187–194.
9. BÖHM, C., JACOPINI, G. Nuove tecniche di programmazione semplificanti la sintesi di macchine universali di Turing. *Rend. Acc. Naz. Lincei {8}, 32*, (June 1962), 913–922.

LETTERS—cont'd from page 323

On 0 and *O*

EDITOR:

In reading the letter by Mr. Turner [On the Confusion Between "0" and "*O*", *Comm. ACM 9*, 1 (Jan. 1966), 35], I notice that his redefinition of the ALGOL ⟨identifier⟩ fails to allow those such as "012ABC", which contain a digit immediately after the zero. It seems that such a combination of characters will pose no major recognition problems, and should be allowed, providing only that it contains a letter somewhere, other than the initial "0", which may logically be taken as a letter or a digit.

I therefore offer an addition to Mr. Turner's redefinition:

⟨identifier⟩ ::= ⟨letter⟩|⟨identifier⟩⟨letter⟩|⟨identifier⟩⟨digit⟩|
0⟨identifier⟩| 0⟨unsigned integer⟩⟨letter⟩

I personally have avoided most of the confusion between the two characters by attempting never to use either character in mnemonic symbols unless it has some mnemonic significance and it is difficult to take it to be the other. There are, however, some situations which are beyond the control of the programmer, but with which we are made to live; two of these are the FORTRAN internal functions, MAX0F and MIN0F. For situations such as these, Mr. Turner's solution seems somewhat appropriate.

MICHAEL L. PERSHING
Department of Computer Science
University of Illinois
Urbana, Illinois 61803

OPTIMAL CODE FROM FLOW GRAPHS
OR
NOTES ON AVOIDING GOTO STATEMENTS

by

M. V. S. Ramanath and Marvin Solomon

Computer Sciences Technical Report #415

January 1981

Optimal Code from Flow Graphs

or

Notes on Avoiding Goto Statements

by

M. V. S. Ramanath and Marvin Solomon

ABSTRACT

This paper considers the problem of generating a linear sequence of instructions from a flow graph so as to minimize the number of jumps. We show that for programs constructed from atomic statements with semicolon, if-then, if-then-else, and repeat-until, the minimal number of unconditional jumps is bounded from above by e+1 and from below by max{ e-b+1, $\lceil (e+1)/2 \rceil$ }, where e is the number of if-then-else statements and b is the number of repeat-until statements. We show that these bounds are tight and present a linear-time algorithm for finding the optimal translation of a flow graph.

Optimal Code from Flow Graphs

1. INTRODUCTION

Over the years, there has been considerable research in the area called "code optimization", which concerns itself with techniques for producing the best possible machine code from a high-level program. There are many possible definitions of "best possible", and the techniques are highly influenced by the natures of the source language and the target machine. In any realistic situation, the problem of producing optimal code is intractable, so researchers content themselves with producing good but not necessarily optimal code, or code that is optimal with respect to some restricted set of transformations or source programs.

The general class of "global" optimizations includes techniques for re-organizing the flow graph of a program, for example removing invariant expressions from loops. However, surprisingly little attention has been paid to the problem of mapping the resulting flow graph into the linear form required by most machine architectures. Careful attention to this step can result in substantial improvements in both space and time.

For example, consider the programs H_n, defined recursively on n as follows:

$$H_\emptyset = S_\emptyset$$

$$H_n = \begin{cases} \underline{\text{if}}\ B_n\ \underline{\text{then}}\ H_{n-1}\ \underline{\text{else}}\ S_n & (n \text{ even}) \\ \underline{\text{repeat}}\ \underline{\text{if}}\ B_n\ \underline{\text{then}}\ H_{n-1}\ \underline{\text{else}}\ S_n\ \underline{\text{until}}\ C_n & (n \text{ odd}) \end{cases}$$

(For each i, S_i is some atomic statement and B_i and C_i are Boolean expressions.) Figure 1 shows the flow graph of H_6. (Some nodes are labeled for future reference.) Standard techniques of code generation would translate H_n into the program 'P_n ; exit', where P_i is defined recursively by

	$\underline{\text{if}}\ \underline{\text{not}}\ B_i\ \underline{\text{then}}\ L_i$	N_i: $\underline{\text{if}}\ \underline{\text{not}}\ B_i\ \underline{\text{then}}\ L_i$
	P_{i-1}	P_{i-1}
$S_\emptyset$	$\underline{\text{goto}}\ M_i$	$\underline{\text{goto}}\ M_i$
L_i: S_i		L_i: S_i
M_i:		M_i: $\underline{\text{if}}\ \underline{\text{not}}\ C_i\ \underline{\text{then}}\ N_i$
if i = ∅	if i > ∅ and i is even	if i > ∅ and i is odd

The translation of H_6 is shown in Figure 2a. A more sophisticated code generator would produce "goto M3" instead of "goto M2" and "goto M5" instead of "goto M4", but a much better translation is T_n, where T_n is

	P_{n-1}
	L_n: $\underline{\text{if}}$ Bn $\underline{\text{then}}\ L_{n-1}$
	S_n
P_n	$\underline{\text{goto}}\ M_{n-1}$
M_{n+2}: $\underline{\text{exit}}$	M_{n+1}: $\underline{\text{exit}}$
n odd	n even

P_i is defined by

L_{i-1}: <u>if</u> B_{i-1} <u>then</u> L_{i-2}
S_{i-1}

$L_\emptyset$: $S_\emptyset$
M_1: <u>if</u> C_1 <u>then</u> M_3
L_1: <u>if</u> B_1 <u>then</u> $L_\emptyset$
S_1
<u>goto</u> M_1

M_i: <u>if</u> C_i <u>then</u> M_{i+2}
L_i: <u>if</u> B_i <u>then</u> L_{i-1}
S_i
<u>goto</u> M_i

$i = 1$ $\qquad$ $i > 1$, i odd

and the initial entrance to H_n is at L_n. The translation of H_6 according to this scheme is shown in Figure 2b. There are n jumps in the first translation of H_n and only n/2 in the second.

In this paper, we confine our attention to translations that preserve the topology of the flow graph exactly, and ignore improvements that might result from techniques such as node splitting or loop unrolling [1]. Under this restriction, there is a one-to-one correspondence between nodes in the graph and instructions other than jumps in the translation. An optimal translation is thus one that minimizes the number of jumps. Since each goto-free segment of the translation corresponds to a simple path in the flow graph, the problem reduces to finding a partition of the graph into as few disjoint simple paths as possible.

A jump-free translation is possible if and only if the graph has a Hamiltonian path. The Hamiltonian path problem is known to be NP-complete, even for planar graphs with in-degree and out-degree bounded by 2 [2]. Since NP-complete problems are widely conjectured to require exponential time for their solution, we do not try to find optimal translations for <u>arbitrary</u> flow graphs, but restrict our attention to "structured" flow graphs that arise

from programs composed of if-then-else, if-then, and repeat-until statements.

The remainder of this paper is organized as follows: Section 2 sketches the definitions and formally states the basic problem. Section 3 presents a linear-time algorithm for finding the optimal translation of any program that uses only if-then-else and repeat-until statements. Section 4 states and proves bounds on the the cost of a partition and proves that the algorithm finds a optimal partition. Section 5 shows that the bounds are tight by exhibiting families of graphs for which the cost of an optimal partition attains the upper and lower bounds. Section 6 shows how to accommodate if-then statements (without an else clause). Section 7 compares our work to previous results and indicates the direction of our current research.

2. DEFINITIONS

We assume the reader is familiar with standard terms of graph theory such as directed graph (digraph), directed acyclic graph (DAG), node, arc, and simple path. By "path" we will mean "simple path".

A flow graph is a digraph G = (N,A) together with a distinguished start node s(G) and set EX(G) of exit nodes, such that each node is reachable from the start node. A simple flow graph (SFG) is a flow graph constructed according to the following rules:

1. A single node n is an SFG with s = n and EX = {n} .

2. If $P = (N_P, A_P)$ and $Q = (N_Q, P_Q)$ are SFG's then a new SFG $T = (N_T, A_T)$ may be constructed from P and Q by any of the following four operations (see Figure 3):

CAT (write T = C(P,Q)): $N_T = N_P \cup N_Q$;
$A_T = A_P \cup A_Q \cup \{(x,s(Q)) \mid x \in EX(P)\}$; $s(T) = s(P)$;
$EX(T) = EX(Q)$.

IF (write T = I(P,i)): Let i be a new node. Then
$N_T = N_P \cup \{i\}$; $A_T = A_P \cup \{(i,s(P)\}$; $s(T) = i$;
$EX(T) = EX(P) \cup \{i\}$.

ELSE (write T = E(P,Q,i)): Let i be a new node. Then
$N_T = N_P \cup N_Q \cup \{i\}$; $A_T = A_P \cup A_Q \cup \{(i,s(P)), (i,s(Q))\}$;
$s(T) = i$; $EX(T) = EX(P) \cup EX(Q)$.

REPEAT (write T = R(P,t): Let t be a new node. Then
$N_T = N_P \cup \{t\}$; $A_T = A_P \cup \{(t,s(P)\} \cup \{(x,t) \mid x \in EX(P)\}$;
$s(T) = s(P)$; $EX(T) = \{t\}$.

The number of applications of ELSE is called the branching factor of G, denoted e(G). The back arcs of G (denoted B(G)) are the arcs of the form (t,s(P)) introduced by REPEAT; the scope of the back arc (t,s(P)), denoted SCOPE(t,s(P)) is N_P. The number of back arcs is denoted b(G). It should be clear that every SFG is reducible [3,4] and that the set B(G) is precisely the unique set of back-arcs [3]. Hence B(G) and SCOPE(a), for each $a \in B(G)$, are independent of the construction of G. The branching factor

is also an inherent property of G.

A restricted SFG is one constructed without any use of IF. If G be an SFG, the restricted SFG corresponding to G is the SFG obtained by replacing each use of I(P,i) in the construction of G with C({i},P).

Assume G is a restricted SFG.

The set A_G - B(G) is called the set of DAG edges of G. A partition p of G is a set of simple paths such that each node of G is in exactly one path. A path using only DAG edges is a DAG path; a DAG partition is one composed of DAG paths. The cost of the partition, c(p), is the number of paths in it. The cost of G, c(G), is the cost of a cheapest partition of G. Partition p is optimal if c(p) = c(G).

A path is a top hook if it starts at s(G) and a bottom hook if it ends at a node in EX(G). A partition is top-open if it contains a top hook, bottom-open if it contains a bottom hook, open if it contains both a top hook and a bottom hook, and nice if it contains a top hook and a bottom hook that are distinct.

The algorithm for finding an optimal partition of G produces two partitions for each subgraph in the construction of G; one is an optimal partition and the other is an optimal open partition. The next definition is used in building these partitions of a graph from partitions of its parts.

Let P and Q be restricted SFG's, and let p_P and p_Q be partitions of them (see Figure 4.)

(CAT) If T = C(P,Q), define the partition $PC(p_P,p_Q)$ of T as follows: If p_P is bottom-open and p_Q is top-open, let h_P be a bottom hook of p_P (distinct from the top hook if possible) and h_Q be the top hook of p_Q. Then $PC(p_P,p_Q) = \{h_P\ h_Q\} \cup (p_P - \{h_P\}) \cup (p_Q - \{h_Q\})$. Otherwise $PC(p_P,p_Q) = p_P \cup p_Q$.

(ELSE) If T = E(P,Q,i) and at least one of p_P, p_Q is top-open, define the partition $PE(p_P,p_Q,i)$ of T as follows: If p_P is top-open, let h_P be its top hook. Then $PE(p_P,p_Q,i) = (i\ h_P) \cup (p_P - \{h_P\}) \cup p_Q$. Similarly, if p_P is not top-open, but p_Q is, $PE(p_P,p_Q,i) = (i\ h_Q) \cup (p_Q - \{h_Q\}) \cup p_P$.

(REPEAT) If T = R(P,t) and p_P is open, define partitions $PR(p_P,t)$ and $PR'(p_P,t)$ of T as follows: Let h_t and h_b be top and bottom hooks of p_P with $h_t \neq h_b$ if p_P is nice. Then $PR'(p_P,t) = \{h_b\ t\} \cup (p_P - \{h_b\})$ and $PR(p_P,t) = \{h_b\ t\ h_t\} \cup (p_P - \{h_b,h_t\})$ if p_P is nice and $PR(p_P,t) = PR'(p_P,t)$ otherwise.

3. THE ALGORITHM

We are now ready to state the main algorithm of this paper:

3.1 Algorithm PARTITION

Input. A restricted SFG G.

Output. Two partitions p and p' for G.

Method. If G = C(P,Q), call PARTITION recursively to get partitions p_P and p'_P for P and partitions p_Q and p'_Q for Q. Let $p' = PC(p'_P,p'_Q)$. Let $p = p'$ if either $c(p'_P) = c(p_P)$ or $c(p'_Q) = c(p_Q)$, and let $p = PC(p_P,p_Q)$ otherwise.

If G = E(P,Q,i), call PARTITION recursively to get partitions p_P and p'_P for P and partitions p_Q and p'_Q for Q. Let $p = PE(p_P,p'_Q)$ if $c(p'_Q) = c(p_Q)$ but $c(p'_P) \neq c(p_P)$. Otherwise, let $p = PE(p'_P,p_Q)$. Let $p' = p$.

If G = R(P,t), call PARTITION recursively to get partitions p_P and p'_P for P. Let $p' = PR'(p'_P)$, and let $p = PR(p'_P)$ if p'_P is nice; let p = p' otherwise.

3.2 Theorem

The partitions p and p' computed for G from Algorithm 3.1 have the following properties:

1. p is optimal.
2. $c(p') \leq c(p) + 1$.
3. p' is open.

4. If p' is not optimal then p' is nice and no optimal partition of G is top-open or bottom-open.

5. If G has a nice partition of cost c(p'), then p' is nice.

Proof. The proof is by induction on the construction of G.

The result is trivial if G is the one-node graph.

If G = C(P,Q), four cases arise:

Case I. p'_P and p'_Q are both optimal. By definition, $p'_G = p_G$ and $c(p_G) = c(P) + c(Q) - 1$. If we could bet a cheaper partition for G, we would be able to decompose it into partitions for P and Q, one of which must be better than optimal. Thus property 1 is proved. Properties 2, 3, and 4 are easy. Property 5 follows from the fact that a nice optimal partition for G can be decomposed into partitions for P and Q, one of which must be optimal and nice. Hence, using 5 inductively, either p'_P or p'_Q is nice and so is p'_G.

Case II. p'_P is not optimal, but p'_Q is optimal. Here too, $p'_G = p_G$ by definition, and $c(p_G) = c(P) + c(Q)$. Properties 2 and 3 are obvious. Using 4 inductively, we see that p'_P is nice and hence so is p'_G. So property 5 is proved. To prove 1, we note that any partition cheaper than p'_G can be used to yield an optimal partition for P whcih is bottom-open, violating property 4 for P. Property 4 follows from 1.

Case III. p'_P is optimal, but p'_Q is not optimal. This case is very similar to case II.

Case IV. Both p'_P and p'_Q are suboptimal. From the definition, we

see that $c(p'_G) = c(P) + c(Q) + 1$. By property 4, neither p_P nor p_Q is open at either end, so $c(p_G) = c(P) + c(Q)$. Also, by an inductive use of 4, p'_P and p'_Q are nice and hence so is p'_G. Thus 2, 3, and 5 are proved. To prove 4, suppose an optimal partition of G were top-open or bottom open. We could then get a top-open partition that is optimal for P or for Q, violating property 4 for P or for Q. Property 1 is proved as in case II.

If $G = E(P,Q,i)$, we have the same four cases as for CAT. In all cases $p'_G = p_G$ and so properties 2 and 3 are obvious and 4 follows from 1. Thus, only 1 and 5 need proof.

Case I. p'_P and p'_Q are both optimal. Here $c(p'_G) = c(P) + c(Q)$ and p'_G is nice. Thus 5 is proved. Property 1 follows from the fact that any partition for G better than p'_G can be used to produce a better-than-optimal partition for P or for Q.

Case II. p'_P is not optimal, but p'_Q is optimal. Here $c(p'_G) = c(P) + c(Q)$. Property 1 follows as in Case I. To prove 5, suppose p'_G is not nice. Then p'_Q is not nice. An inductive use of 4 shows that any optimal nice partition for G yields an optimal nice partition for Q, which is a contradiction, since p'_Q is not nice.

Case III. p'_P is optimal, but p'_Q is not. The proof is similar to case II.

Case IV. p'_P and p'_Q are both suboptimal. Here p'_P is nice by property 4 so p'_G is nice, proving property 5, and $c(p'_G) = c(P) + 1$. Property 1 follows from the fact that any partition better than

p'_G would yield an optimal top-open partition for P or for Q, violating property 4 of the inductive hypothesis.

Finally, we consider the case that G = R(P,t). Properties 2 and 3 are obvious. We have three cases:

Case I. p'_P is optimal and nice. Clearly, p'_G is nice, proving property 5. Since $c(p_G) = c(p_P) - 1$, any partition for G better than p_G would yield a partition for P better than optimal. Hence property 5 is proved. The proof of 4 is similar.

Case II. p'_P is optimal but not nice. In this case, $c(p_G) = c(p'_G) = c(p_P)$. Property 4 follows from 1, which may be proved by arguments similar to case I above. Property 5 follows from the fact that an optimal nice partition for G would imply an optimal nice partition for P.

Case III. p'_P is not optimal. In this case $c(p'_G) = c(p_P) + 1 = c(P) + 1$ and $c(p_G) = c(P)$. p'_P is nice so p'_G is nice and 5 is proved. Properties 1 and 4 follow by the usual arguments.

This conpletes the proof of Theorem 3.2. The algorithm is clearly linear in the length of the derivation of G, and hence in the size of G.

4. BOUNDS ON COSTS

In this section, we derive upper and lower bounds on the cost of a restricted SFG.

4.1 Theorem

Let G be a restricted SFG. Then

$$\max \{ e(G)-b(G)+1, \lceil (e(G)+1)/2 \rceil \} \leq c(G) \leq e(G) + 1$$

Before proving 4.1 we state and prove some preliminary results.

4.2 Lemma

If p is any DAG partition of G, then $c(p) \geq e(G) + 1$; there is an algorithm to find a DAG partition such that $c(p) = e(G) + 1$.

Proof The usual code-generation algorithm produces a partition of cost $e(G) + 1$. The proof that this cost is the best possible is by induction on the construction of G.

If G is a single node, the result is trivial. Otherwise, let p_G be a DAG partition of G.

If $G = C(P,Q)$, then p_G clearly decomposes into DAG partions p_P and p_Q of P and Q, respectively, such that $c(p_G) \geq c(p_P) + c(p_Q) - 1$. By the induction hypothesis, $c(p_P) \geq e(P) + 1$ and $c(p_Q) \geq e(Q) + 1$, so $c(p_P) \geq e(P) + 1 + e(Q) + 1 - 1 = e(P) + e(Q) + 1 = e(G) + 1$.

If $G = E(P,Q,i)$, then $e(G) = e(P) + e(Q) + 1$ and p_P can be decomposed into DAG partitions of P and Q such that $c(p_G) \geq c(p_P) + c(p_Q)$. Once again, by the inductive hypothesis, $c(p_P) \geq c(p_P) + c(p_Q) \geq e(P)+1+e(Q)+1 = e(G) + 1$.

If $G = R(P,t)$, then there is a DAG partition of P such that $c(p_G) \geq c(p_P)$. By induction, $c(p_G) \geq c(p_P) \geq e(P) + 1 = e(G) + 1$.

This proves lemma 4.2.

4.3 Corollary

For any partition p_G of an SFG G,

(i) $c(p_G) \geq e(G) - b(p_G) + 1$

(ii) $c(p_G) \geq \lceil (e(G)+1)/2 \rceil$

(iii) if $c(p_G) = \lceil (e(G)+1)/2 \rceil$, then $\lfloor (e(G)+1)/2 \rfloor \leq b(p_G) \leq \lceil (e(G)+1)/2 \rceil$

Proof Deleting back arcs from p_G yeilds a DAG partition p'_G of cost $c(p_G) + b(p_G)$. Hence, by Lemma 4.2, $c(p_G) + b(p_G) \geq e(G) + 1$, and (i) follows. To prove (ii), suppose $c(p_G) \leq \lceil (e(G)+1)/2 \rceil$. Then $b(p_G) \leq c(p_G) < \lceil (e(G)+1)/2 \rceil$, so by 4.2, $\lceil (e(G)+1)/2 \rceil + \lfloor (e(G)+1)/2 \rfloor = e+1 \leq c(p'_G) = c(p_G) + b(p_G) < \lceil (e(G)+1)/2 \rceil + b(p_G) < \lceil (e(G)+1)/2 \rceil + \lceil (e(G)+1)/2 \rceil$. Cancelling occurences of $\lceil (e(G)+1)/2 \rceil$ yeilds $\lfloor (e(G)+1)/2 \rfloor < b(p_G) < \lceil (e(G)+1)/2 \rceil$, which is impossible.

The proof of part (iii) is the same as part (ii), except all occurences of $<$ should be replaced by $\leq$.

Proof of Theorem 4.1. The upper bound follows directly from the Lemma 4.2. One lower bound follows directly from 4.3(ii). The other lower bound is proved inductively:

If G is a single node, then $e(G)-b(G)+1 = 1 = c(G)$.

If $G = C(P,Q)$, then $c(G) \geq c(P)+c(Q)-1 \geq (e(G)-b(G)+1) + (e(Q)-b(Q)+1) - 1 = (e(P)+e(Q)) - (b(P)+b(Q)) + 1 = e(G)-b(G)+1$.

If $G = E(P,Q,i)$, then $c(G) \geq c(P)+c(Q) \geq (e(G)-b(G)+1) + (e(Q)-b(Q)+1) = (e(P)+e(Q)+1) - (b(P)+b(Q)) + 1 = e(G)-b(G)+1$.

If $G = R(P,t)$, then $c(G) \geq c(P)-1 \geq (e(G)-b(G)+1) - 1 = e(P) - (b(P)+1) + 1 = e(G)-b(G)+1$.

5. ADDING IF-THEN STATEMENTS

In this section we show that the results for restricted SFG's remain valid when if-then statements are added. Intuitively, the construction "if B then S" is modelled by "if B then S else skip". However, rather than introduce skip as a primitive concept, we model the if-then statement as C(i,P) (where i is a new node representing the condition B and P is the flow graph of S), and make i an additional exit node.

5.1 Theorem

Let G be an SFG and G' the corresponding restricted SFG. Any optimal partition p of G can be effectively transformed into a partition p' of G' such that $c(p') \leq c(p)$.

Proof (sketch). Call an arc (i,n) a forward arc if i is the node introduced by the operation T = I(P,i), but n is not s(P) (see Figure 5). G and G' differ only in that forward arcs are present in the former and absent in the latter; hence, to transform p to p', we need only eliminate all forward arcs from p.

Choose an innermost forward arc (i,n) used by p. Since the paths in p are node-disjoint, p does not use the arc (i,s(P)). That arc is the only arc entering the subgraph P, so p can be decomposed into paths outside P and paths inside P. The set of paths inside P forms an optimal partition p_P of P, so by Theorem 3.2, it may be replaced by an open partition p'_P of P with at most one more path. Modify the original partition of G by replacing p_P with p'_P. Then remove the path that uses (i,n), say u(i,n)v, add u to the top hook of p'_P, and add v to a bottom hook of p'_P. (The latter operation is possible since the construction of an SFG ensures that any successor of any exit node of a subgraph is a successor of every exit node of that subgraph. Hence n is a successor of each exit node of P.) This construction deletes the path u(i,n)v, so even if $c(p'_P) = c(p_P)+1$, the net increase in cost is zero.

5.2 Corollary.

If p is an optimal partition for G' then it is also an optimal partition for G.

6. TIGHTNESS OF BOUNDS

In this section, we show that the bounds derived in Section 4 are tight.

6.1 Theorem.

For any positive integer e, there are graphs G_1 and G_2 with branching factor e such that $c(G_1) = e+1$ and $c(G_2) = \lfloor (e+1)/2 \rfloor$. If b is an integer such that $e-b+1 > \lfloor (e+1)/2 \rfloor$, there is also a graph G_3 such that $e(G_3) = e$, $b(G_3) = b$, and $c(G_3) = e-b+1$.

Proof. Let G_1 be any graph with branching factor e and no loops ($b(G_1) = \emptyset$). By Theorem 4.1, $c(G_1) = e+1$.

Let G_2 be the graph H_e defined in the introduction: $H_\emptyset = S_\emptyset$, $H_e = E(H_{e-1}, S_e, B_e)$ if e is even, and $H_e = R(E(H_{e-1}, S_e, B_e))$ if e is odd. Then the partition created by the algorithm is

$$\{B_{2i}S_{2i}C_{2i+1}B_{2i+1}S_{2i+1} \mid 1 \le i \le \lfloor (e-1)/2 \rfloor\} \cup \{S_\emptyset C_1 B_1 S_1,\ B_e S_e\}$$

(The last path mentioned above is omitted if e is odd.)

If $\lceil (e+1)/2 \rceil < e-b+1$, then $b < \lfloor (e+1)/2 \rfloor$. Let G_3 be H_e with all but the b innermost back arcs deleted. The partition of cost e-b+1 is the partition p above, modified by removing the deleted arcs and splitting the paths that contained them in two.

7. SUMMARY AND CONCLUSIONS

Considering the amount of work that has been done on program optimization, it is surprising that more attention has not been paid to the problem tackled in this paper. Most literature on program optimization deals either with transformations on the flow graph of a program or with translation of an individual statement into machine code. The only other work we know of in this area is by Boesch and Gimpel [5]. They show that in the simple case that the flow graph is acyclic, the optimum partition problem can be reduced to the maximum matching problem for bipartite graphs. Hence any good algorithm for maximum matchings, such as the $O(n^{2.5})$ algorithm of Hopcroft and Karp [6], yields an algorithm for optimal partition of an acyclic flow graph. Since acyclic flow graphs are rare in practice, they present a heuristic algorithm for arbitrary graphs that proceeds by performing an interval analysis of the graph [3], finding optimal partition for the intervals, and pasting the partitions together. However, this procedure does not, in general, yield an optimal partition, and Boesch and Gimpel present no results on how close to optimal it comes.

Other related work involves investigations into the effect of long and short branch instructions on code length (see, for example, [7]) and the impact of restricting set of flow graphs to "structured programs" on program efficiency (for example, [8]).

The results here are only preliminary. We are currently extending the methods of this paper to cover other common constructs such as case, while and exit-loop statements. We conjecture that there is a polynomial algorithm for finding an optimal partition of any reducible flow graph.

8. REFERENCES

[1] F. Baskett, "The best simple code generation technique for WHILE, FOR, and DO loops," Sigplan Notices 13, 4, pp. 31-32 (April 1978).

[2] J. Plesnik, "The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree bound 2," Information Proc. Letters 8, 4, pp. 199-201 (April 1979).

[3] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," Journal of the ACM 21, 3, pp. 367-375 (1974).

[4] M. S. Hecht, Flow Analysis of Computer Programs, American Elsevier, New York (1977).

[5] F. T. Boesch and J. F. Gimpel, "Covering the points of a digraph with point-disjoint paths and its application to code optimization," Journal of the ACM 24, 2, pp. 192-198 (April 1977).

[6] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs.," SIAM Journal on Computing 2, 4, pp. 225-230 (December 1973).

[7] T. G. Szymanski, "Assembling code for machines with span-dependent instructions," CACM 21, 5, pp. 300-308 (April 1978).

[8] R. A. DeMillo, S. C. Eisenstat, and R. J. Lipton, "Can structured programs be efficient?," SIGPLAN Notices, pp. 10-18 (October 1976).

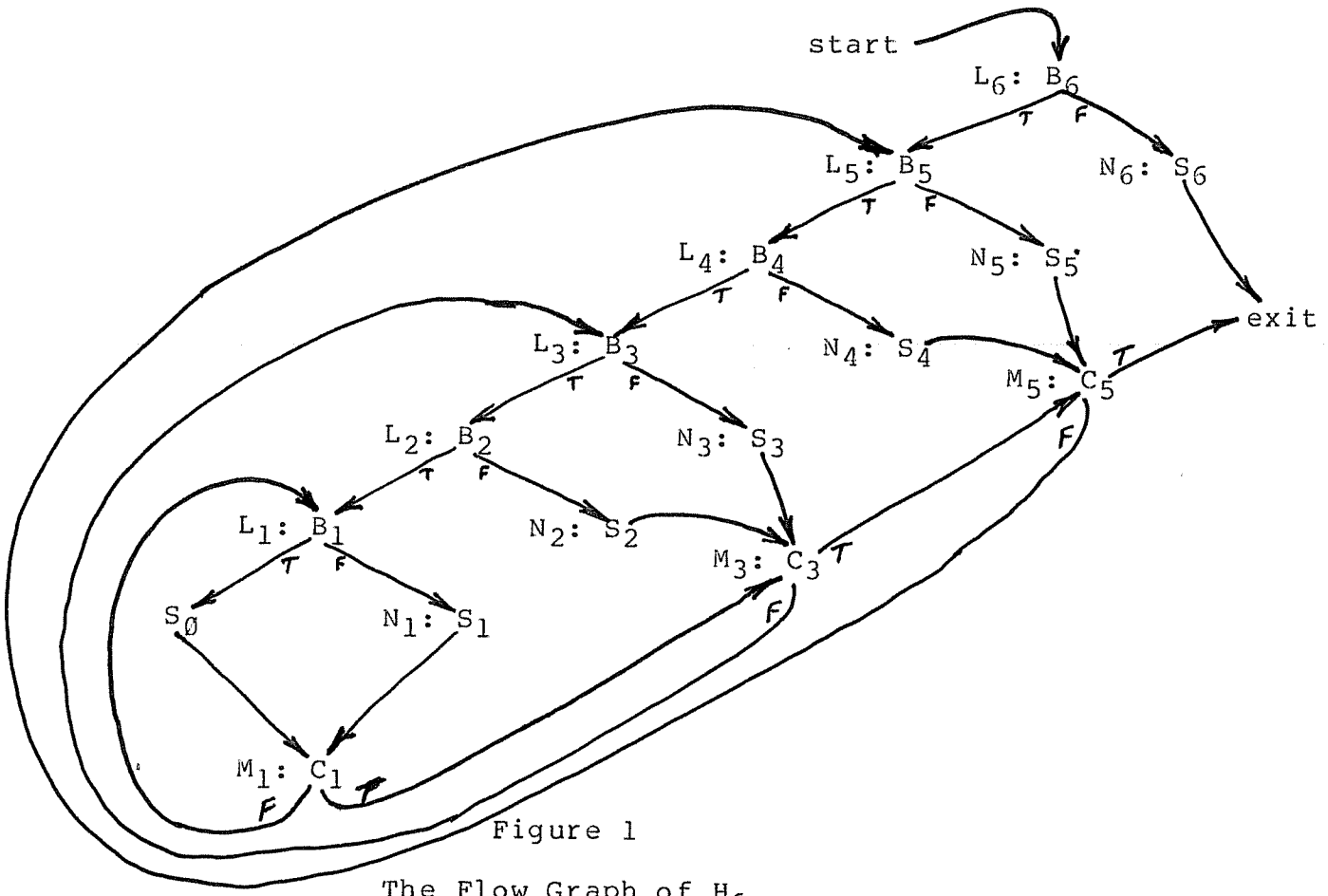


Figure 1

The Flow Graph of H_6

```
start:
    if not B6 then N6
L5: if not B5 then N5
    if not B4 then N4
L3: if not B3 then N3
    if not B2 then N2
L1: if not B1 then N1
    Sø
    goto M1
N1: S1
M1: if not C1 then L1
    goto M2
N2: S2
M2: goto M3
N3: S3
M3: if not C3 then L3
    goto M4
N4: S4
M4: goto M5
N5: S5
M5: if not C5 then L5
    goto M6
N6: S6
M6: exit
```

(a)

```
Lø: Sø
M1: if C1 then M3
L1: if B1 then Lø
    S1
    goto M1
L2: if B2 then L1
    S2
M3: if C3 then M5
L3: if B3 then L2
    S3
    goto M3
L4: if B4 then L3
    S4
M5: if C5 then M7
L5: if B5 then L4
    S5
    goto M5
start:
L6: if B6 then L5
    S6
M7: exit
```

(b)

Figure 2

Two Translations of H_6

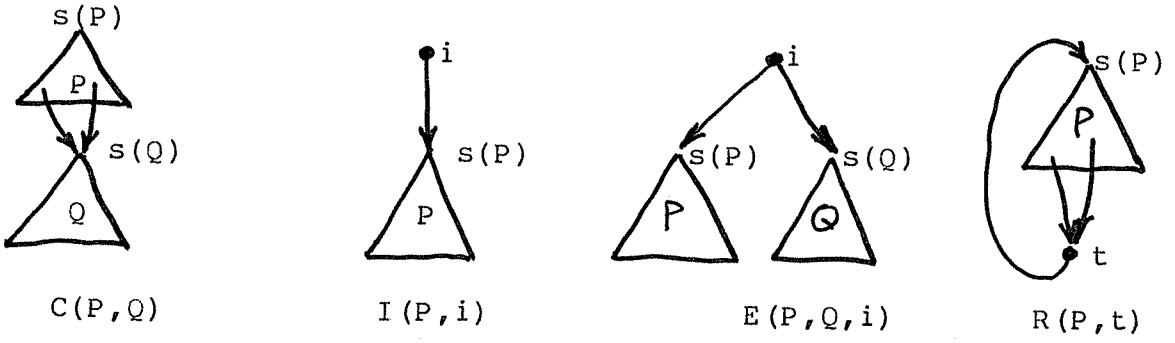


Figure 3

SFG Operations

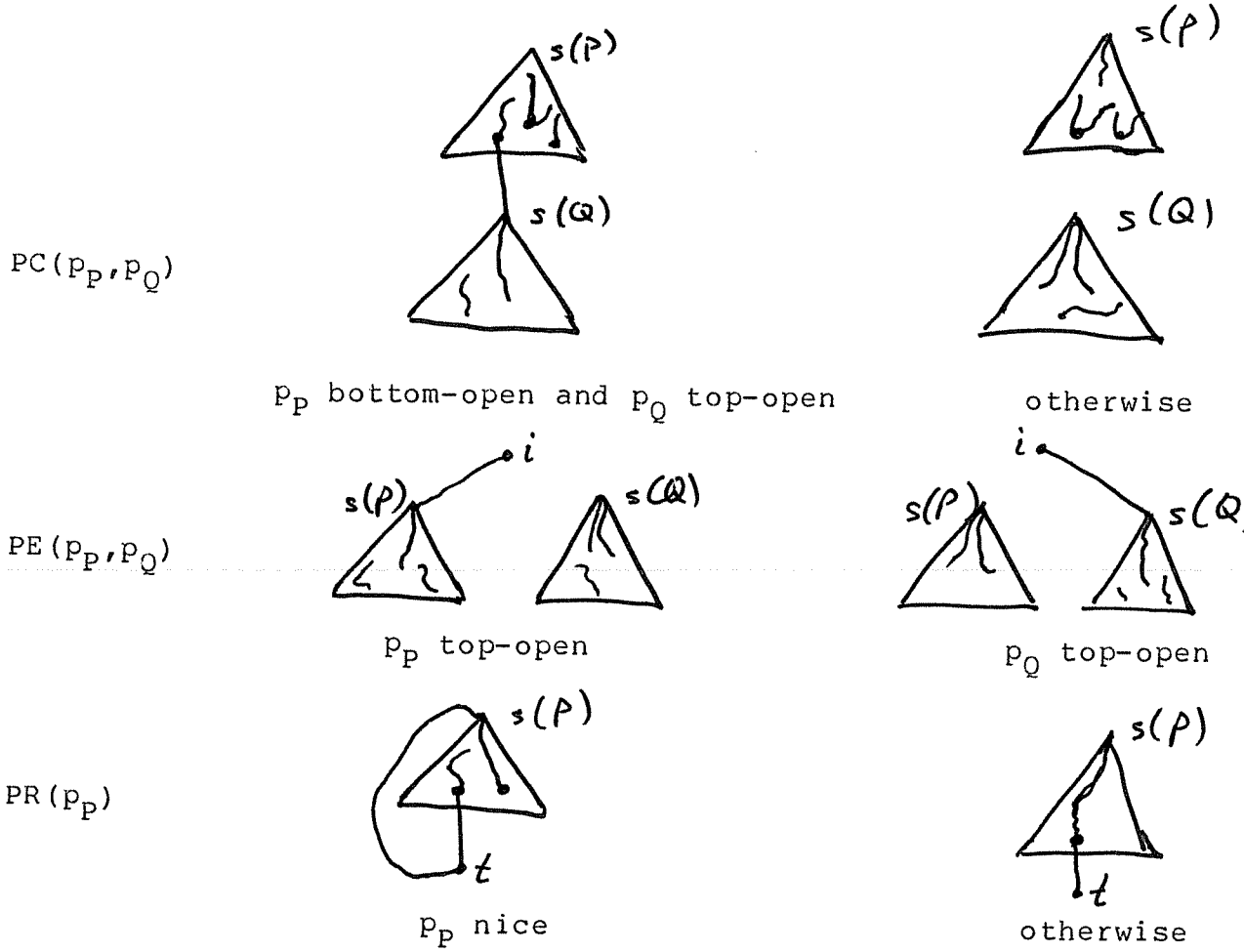


Figure 4

Operations for Combining Partitions

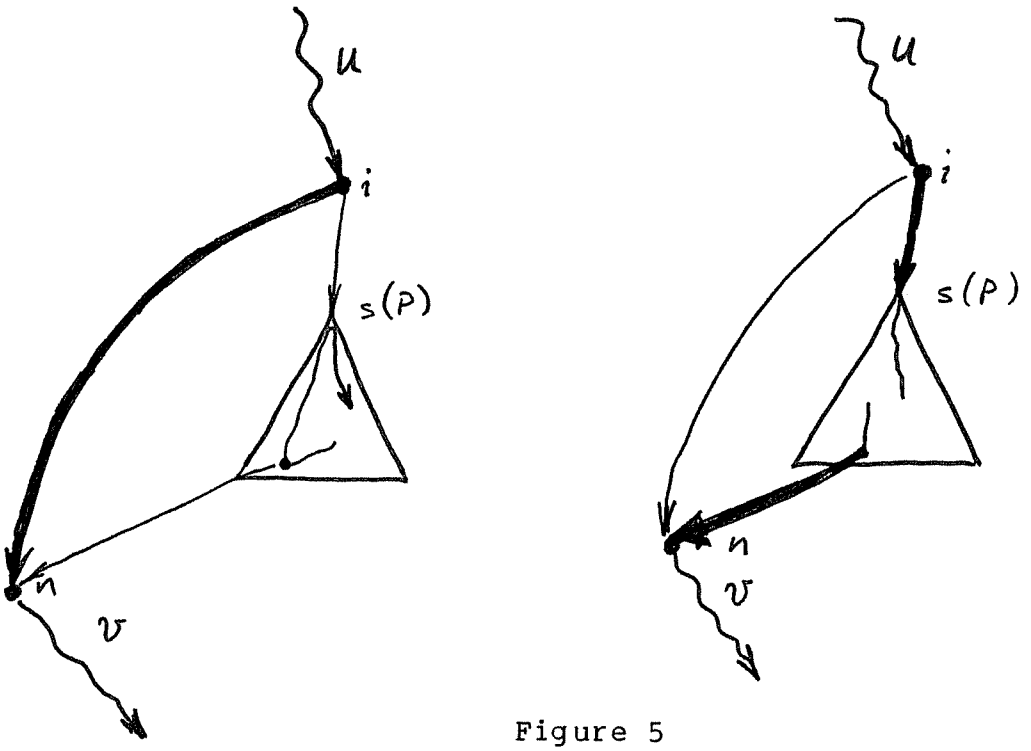


Figure 5

Eliminating Forward Arcs

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 60 (2006) 82–116
Science of Computer Programming
www.elsevier.com/locate/scico

To use or not to use the **goto** statement: Programming styles viewed from Hoare Logic

Hidetaka Kondoh[a,*], Kokichi Futatsugi[b]

[a] *Systems Development Lab., Hitachi, Ltd., 1099 Ohzenji, Asao, Kawasaki, Kanagawa 215-0013, Japan*
[b] *JAIST, 1-1 Asahi-dai, Tatsunokuchi-machi, Noumi-gun, Ishikawa 923-1292, Japan*

Received 11 November 2003; received in revised form 28 April 2005; accepted 18 May 2005
Available online 31 August 2005

Abstract

There has been a vast amount of debate on the **goto** issue: i.e., the issue whether to use or not to use the **goto** statement initiated by Dijkstra in his famous Letter to the Editor of CACM and his proposal of 'Structured Programming'. However, except for the **goto**-less programming style by Mills based on theoretical results on the expressibility of control flow diagrams, there have hardly been any scientific accounts on this issue from Dijkstra's own viewpoint of the correctness of programs. In this work, we reconsider this seemingly old-tired issue from the viewpoint of Hoare Logic, the most well-known framework for correctness proof of programs. We show that, in two cases, the with-**goto** programming styles are more suitable for proving correctness in Hoare Logic than the corresponding without-**goto** ones; that is, in each of two cases, the without-**goto** style requires more complicated assertions in the proof-outline than the with-**goto** one. The first case is on the use of the **goto** statement for escaping from nested loops and the second case is on the use of the **goto** statement for expressing state transitions in programming through the finite state machine model. Hence, in both cases, the use of the **goto** statement can be justified from the viewpoint of the correctness proof in Hoare Logic.

Keywords: Structured programming; Verification pattern; Hoare logic; Finite state modeling

* Corresponding author.

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.05.003

1. Introduction

The themes of this work are the so-called **goto** controversy and structured programming; both are very classical and may seem to be too old-tired to most readers, and hence some explanation is necessary why such classical themes have to be discussed now.

Our motivation for discussing these themes in the present work is that these themes, in our opinion, have not been studied sufficiently along the direction which Dijkstra originally expected; i.e., these themes have hardly been studied scientifically.

In order to clarify this point, we summarize the history on these themes. Both themes firmly relate the issue of whether to use or not to use (and, if to use, how to use) the **goto** statement. So, hereafter, we call this issue *'the* **goto** *issue'* for short and this **goto** issue is the theme of the present work. Note that, by the term '**goto** statement' (or simply '**goto**'), we mean any kind of jump, including limited ones, such as **exit** (in Ada) and **break** (in C), but not including exception handling. This is because exception handling is far from a simple jump and is a far more complicated (and powerful) language mechanism.

Needless to say, writing understandable programs is a very important issue in programming methodology and software engineering. In his famous Letter to the Editor of CACM [15], Dijkstra warned of the negative effects of the **goto** statement against the understandability of programs. Consecutively, in his proposal of the concept of 'Structured Programming' [16,17], he wrote (in [17, p. 41]) *"the need for careful program structuring has been put forward as a consequence of the requirement that program correctness can be proved"*. Here, Dijkstra clarified the criterion of *good programs* from rather ambiguous and subjective 'understandability' to 'easiness of correctness proof' which is more suitable for scientific analysis. It should be noted that, in his initial letter, Dijkstra already pointed out that mechanical elimination of **goto** on the basis of Böhm and Jacopini's work [9] is useless in improving the understandability of programs.

Dijkstra's letter caused the so-called **goto** controversy. The report "The GO TO Controversy — Debate" [39] of a meeting held by ACM SIGPLAN tells us how important the **goto** issue was in the study on programming methodology and software engineering throughout the 1970s. How important the **goto** issue was at that time can also be understood from the fact that anthologies [7,25,58–61] on programming methodology and/or on software engineering published during the late 1970s and the early 1980s reprinted many works on this issue.

As we can see from the papers in those anthologies, almost all opinions and works appeared during the **goto** controversy were not studied with the measure on 'easiness of correctness proof' which Dijkstra originally intended. Instead most of them were rather intuitive and/or empirical; cf. a typical opinion in the case for **goto** was [32] and the one in the case against **goto** was [64], and both were intuitive, empirical, and/or emotional.

A rarely exceptional work with theoretical accounts on the **goto** issue was the one by Knuth [36], in which he proposed structured programming using **goto** statements. In that paper, he showed several uses of **goto**s without sacrificing the understandability of programs. Knuth also discussed extended control structures including the so-called $n\frac{1}{2}$ loop from the viewpoint of program verification and gave inference rules of Hoare Logic for those extended control structures which are not popular in practically used programming languages (but are constructible using **goto**s). This work of Knuth, however, still remained

at a rather small scale, namely control structures in programming languages, and did not address more macro scale *patterns* of uses of the **goto** statement to give better programming styles in order to make correctness proofs easier.

Another exceptional study with clear theoretical background on the **goto** issue was given by Mills and his colleagues in their series of works [40,41,43–45,47] which later led to a software development method: namely, Cleanroom Software Engineering [8,22,48,50,51,55].

Mills studied programming styles without **goto** and used the term 'Structured Programming' in a quite different meaning from Dijkstra's original intention: i.e., Mills claimed

$$\text{structured programming} = \textbf{goto}\text{-less programming}.$$

Mills' idea behind his claim was based on results about the expressibility of control flow diagrams. That is, any control flow in sequential programs can be expressed with only three basic control structures: sequencing, conditional and indefinite loop; hence any sequential control flow can be expressed without **goto** statements. On the expressibility of control flows, there are many theoretical works [2,12,13,37] aside from the work by Böhm and Jacopini [9] mentioned earlier.

However, as we noted before, Dijkstra claimed that such theoretical results on the expressibility of control flows could not justify the elimination of **goto** for writing understandable programs. Therefore, when Mills and his colleagues regarded structured programming as **goto**-less programming and proposed their software engineering development method, Dijkstra's own concept of 'structured programming' was drastically changed.

There are many scientific works on the **goto** statement as a programming languages construct. We mention only a few representative works other than those on the expressibility of control flows: on denotational semantics of **goto**, Strachey and Wadsworth [56] invented the notion of continuation; on axiomatic semantics of **goto**, de Bruin [10] as well as Clint and Hoare [11] gave the verification rule for **goto** in Hoare Logic.

Those scientific works, however, only address theoretical properties of the **goto** statement itself. They do not directly give any insight for proper uses of **goto** for well disciplined programming, which Dijkstra was interested in and also which we are going to consider in the present work.

The above is a short historical survey of the **goto** controversy and structured programming. As we can see from this survey, there have hardly been any studies on the **goto** issue from the correctness proof viewpoint, i.e. from the viewpoint of programming logics. This is the reason why we reconsider such classical topics of the **goto** controversy and structured programming now.

In this work, we adopt Hoare Logic as the programming logic with which we investigate the **goto** issue. Then, we show that, in two cases at least, programming styles with **goto**s can have their correctness more naturally proved than those without **goto**s. One case is escaping from nested loops. This is the case for which Dijkstra [15] originally claimed — although Dijkstra carefully credited and attributed the originality of the claim to Hoare — that the use of **goto** could be justified. The other case is for programs whose design is given as a finite state machine. Plauger [52] intuitively claimed that **goto**s were to be used for this case. We will show that Hoare Logic can give scientific accounts to Plauger's intuitive claim.

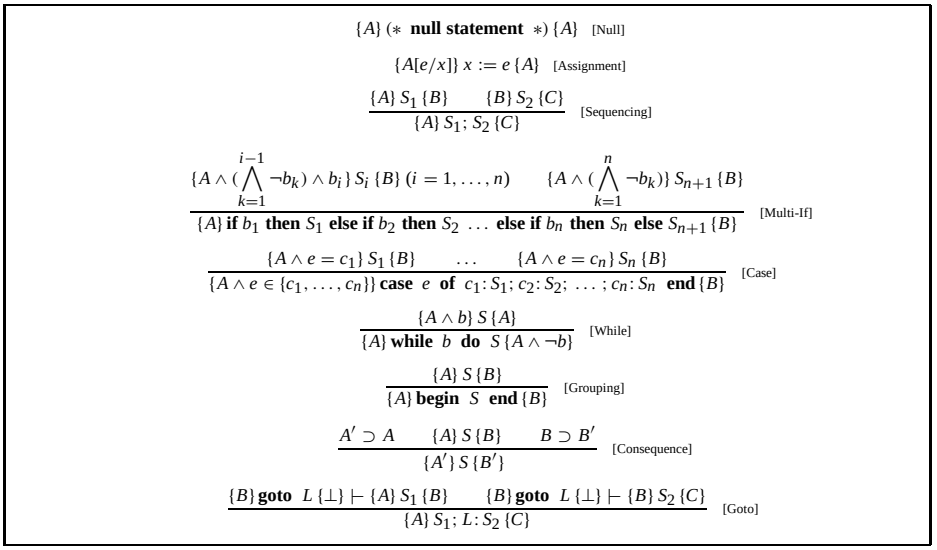


Fig. 1. Axioms and inference rules of Hoare Logic.

The organization of this paper is as follows: Section 2 summarizes the system of Hoare Logic for a fragment of Pascal in which all examples are described. Section 3 discusses the first case, using **goto** in escaping from nested loops. We first discuss the theme on the basis of the correctness proof of a pair of example programs with and without a **goto**, then the discussion is generalized to the level of program schemes. Section 4 considers the latter case, the use of **goto** in programming through the finite state machine model. Again, the discussion starts with a concrete example after Plauger, then it is generalized to the one for program schemes. Section 5 summarizes our results and gives some perspectives.

2. Hoare Logic for a fragment of Pascal

In this section, we summarize the programming language for presenting examples, basically a fragment of Pascal, and axioms and inference rules of Hoare Logic [29] for this language.

Fig. 1 is the system of Hoare Logic used in this paper. Language constructs used in this paper are those in that figure. For the system of Hoare Logic of the (almost) full-set of Pascal, readers may wish to consult the pioneering work by Hoare and Wirth [31].

Since most of the axioms and rules in Fig. 1 are well-known, we do not give any accounts for them. For a general introduction to Hoare Logic, we refer to de Bakker's encyclopedic monograph [6]. Here we give only a few comments on some of those rules. With respect to the [Multi-If] rule, the construct **if else if** ... **else if else** is regarded as nested two-way branching **if** statements in Pascal. In Fig. 1, however, this is regarded as a single construct (like in Ada and Algol 68) and is directly given its own inference rule. This deviation from Pascal is not essential but is useful for decreasing the number of lines of proof-outlines. As regards the [Goto] rule which is due to Clint and Hoare [11], this rule tells us a moral on using the **goto** statement. Premises of this rule

$$\{B\}\,\textbf{goto}\ \ L\,\{\bot\} \vdash \{A\}\,S_1\,\{B\},$$
$$\{B\}\,\textbf{goto}\ \ L\,\{\bot\} \vdash \{B\}\,S_2\,\{C\}$$

means that when we prove Hoare triples $\{A\}\,S_1\,\{B\}$, $\{B\}\,S_2\,\{C\}$ we can assume the triple $\{B\}\,\textbf{goto}\ \ L\,\{\bot\}$ whose precondition is the same as that of S_2 to which this **goto** transfers the control. So the moral of using the **goto** statement is:

Moral *When we use a* **goto** *statement, we must clarify the precondition of the labelled statement to which the* **goto** *jumps.*

File input/output

In Pascal, the notion of file is rather complicated. It is not our intention to discuss detailed correctness proofs of programs with Pascal's file input/output. Hence, in order to prevent such a non-essential complication, we define our notion of file and related concepts in the rest of this section, then we use them throughout this work.

First, we define our notion of finite sequence (hereafter simply called 'sequence' for brevity's sake) and associated basic operations on sequences in order to avoid complication due to the partiality of usual operations on sequences. Hereafter, D denotes the set of elements of sequences. c, d, e are used to denote an element and r, s, t, u runs over the set of all sequences. ε is the constant symbol denoting the empty sequence. eof is the constant symbol for a special value which never appears in sequences: i.e., $\text{eof} \notin D$. Note that the constant symbol eof can be used in program codes in order to write conditional statements for detecting the end of a file.

Decomposing operations for sequences are as follows: $\text{fst}(s)$ and $\text{bwd}(s)$ are the first element and the remaining (backward) part of a sequence s, respectively; when s is an empty sequence ε, $\text{fst}(\varepsilon) \stackrel{\text{def}}{=} \text{eof}$ and $\text{bwd}(\varepsilon) \stackrel{\text{def}}{=} \varepsilon$; $\text{lst}(s)$ and $\text{fwd}(s)$ are the last element and the forward part of s, respectively.

Operations for concatenating/constructing sequences are as follows: $s :: t$ concatenates two sequences s, t into one sequence; $c \cdot\!: s$ constructs a sequence whose first element is c and the backward part is s; $s :\!\cdot\, c$ gives a sequence with c as its last element s as its forward part.

These basic operations for sequences can be used in writing assertions (but not in program codes) and the functionalities of these operations are as follows:

$$\begin{aligned}
\text{fst}, \text{lst} &: D^* \to D \cup \{\text{eof}\},\\
\text{bwd}, \text{fwd} &: D^* \to D^*,\\
(_ :: _) &: D^* \times D^* \to D^*,\\
(_ \cdot\!: _) &: D \times D^* \to D^*,\\
(_ :\!\cdot\, _) &: D^* \times D \to D^*
\end{aligned}$$

where D^* means the set of all finite sequences made from D.

Axioms for these operations are summarized in Fig. 2. In using these operations, parentheses are omitted not only when associative laws in Fig. 2 allows omission but also when they can be uniquely recovered from the functionality of these operations. For example, "$c \cdot\!: d \cdot\!: s$" means not "$(c \cdot\!: d) \cdot\!: s$" but "$c \cdot\!: (d \cdot\!: s)$". When we write "$c \cdot\!: s$" or "$s :\!\cdot\, c$", $c \neq \text{eof}$ must be satisfied due of the functionality of $(\_ \cdot\!: \_)$ or $(\_ :\!\cdot\, \_)$. Hence, when such an expression is used in an assertion, $c \neq \text{eof}$ is not given explicitly.

$$
\begin{gathered}
\mathrm{fst}(\varepsilon) = \mathrm{eof} = \mathrm{lst}(\varepsilon),\\
\mathrm{bwd}(\varepsilon) = \varepsilon = \mathrm{fwd}(\varepsilon),\\
\mathrm{fst}(c \cdot\!: t) = c = \mathrm{lst}(t :\!\cdot c),\\
\mathrm{bwd}(c \cdot\!: t) = t = \mathrm{fwd}(t :\!\cdot c);\\
c \cdot\!: \varepsilon = \varepsilon :\!\cdot c,\\
s :: \varepsilon = s = \varepsilon :: s,\\
c \cdot\!: s \neq \varepsilon \neq s :\!\cdot c;\\
(s :\!\cdot c) :: t = s :: (c \cdot\!: t),\\
(c \cdot\!: s) :\!\cdot d = c \cdot\!: (s :\!\cdot d),\\
(s :: t) :: u = s :: (t :: u),\\
(s :: t) :\!\cdot c = s :: (t :\!\cdot c),\\
(c \cdot\!: s) :: t = c \cdot\!: (s :: t).
\end{gathered}
$$

Fig. 2. Axioms for basic operations on sequences.

We now define our notion of file. A file f consists of two sequences f_{L} and f_{R}: i.e., $f \stackrel{\mathrm{def}}{=} \langle f_{\mathrm{L}}, f_{\mathrm{R}} \rangle$. When f is an input file, f_{L} is the part which have been read already and f_{R} is the part to be read from now on. When f is an output file, f_{L} consists of values already written and f_{R} is always the empty sequence ε. These f_{L} and f_{R} cannot be used in program codes and can appear only in assertions: that is, f_{L} and f_{R} are introduced as specification variables.

Using this notion of file, we define our input/output procedures, get and put, with which we describe example codes in Section 4.

The informal specification of the input procedure get is as follows. Let f be a file and x be a variable. By the invocation get(f, x), the values of x, f_{L} and f_{R} become fst(f_{R}), $f_{\mathrm{L}} :\!\cdot \mathrm{fst}(f_{\mathrm{R}})$ and bwd(f_{R}), respectively. When the whole of f has been read already (i.e., when $f_{\mathrm{R}} = \varepsilon$), the special value eof is assigned to the input variable x.

The informal specification of the output procedure put is as follows. Let f be a file and e be an expression whose value is to be output. The invocation put(f, e) assigns ε to f_{R}. When the value of e is eof, the invocation terminates without adding any new element to f_{L}. When $e \neq \mathrm{eof}$, the value of e is appended at the end of f_{L}.

These input/output procedures can be defined as pseudocodes[1] in Fig. 3. Fig. 4 shows Hoare triples for these input/output procedures. It is easy to check that Hoare triples in Fig. 4 are satisfied by pseudocodes in Fig. 3.

3. Case 1: Escaping from nested loops

In this section, we discuss a typical case usually programmed with the **goto** statement: namely, escaping from nested loops. In fact, many programming languages support limited jump mechanisms for this purpose, such as **exit** in Ada and **break** in C. This is also a very

[1] Those are not real program codes but merely pseudocodes because f_{L} and f_{R} are not program variables but specification ones and basic operations on sequences, allowed to appear only in assertions, are used.

```
get(f, x) =def  if f_R = ε then
                  x := eof
                else
                  begin
                    x := fst(f_R);
                    f_L := f_L :· fst(f_R);
                    f_R := bwd(f_R)
                  end

put(f, e) =def  f_R := ε;
                if e ≠ eof then
                  f_L := f_L :· e
```

Fig. 3. Pseudocodes for input/output procedures.

$\{ f_R = \varepsilon \wedge P[\mathrm{eof}/x] \vee f_R \neq \varepsilon \wedge Q[\mathrm{fst}(f_R)/x, (f_L :\cdot \mathrm{fst}(f_R))/f_L, \mathrm{bwd}(f_R)/f_R] \}$
$\mathrm{get}(f, x)$
$\{x = \mathrm{eof} \wedge f_R = \varepsilon \wedge P \vee x \neq \mathrm{eof} \wedge Q\}$

$\{e = \mathrm{eof} \wedge P[\varepsilon/f_R] \vee e \neq \mathrm{eof} \wedge P[(f_L :\cdot e)/f_L, \varepsilon/f_R]\}$
$\mathrm{put}(f, e)$
$\{P\}$

Fig. 4. Hoare triples for input/output procedures.

basic case for which, in his Letter [15], Dijkstra admitted the use of the **goto** statement to be appropriate: *"I remember having read the explicit recommendation to restrict the use of the* **goto** *statement to alarm exits, . . .; presumably, it is made by C.A.R. Hoare"*.

The purpose of this section is to give, on the basis of Hoare Logic, scientific accounts to this empirically established programming style.

3.1. Example: Finding a 0-element in a two-dimensional array

We consider the following problem:

Example 1. Let a[1..m, 1..n] be a two-dimensional array. If there are 0-elements (elements whose value is 0), then the pair of variables i and j must be assigned the lexicographically minimal pair of indices of such 0-elements. Otherwise, i must be set a value greater than m.

```
1    i := 1;
2    while i ≤ m do
3      begin
4        j := 1;
5        while j ≤ n do
6          if a[i, j] = 0 then
7            goto 99
8          else
9            j := j + 1;
10       i := i + 1
11     end;
12   99: (* the end of the program *)
```

Fig. 5. The with-**goto** program for searching a 0-element in a two-dimensional array.

```
1    i := 1;
2    nYF := true;
3    while i ≤ m and nYF do
4      begin
5        j := 1;
6        while j ≤ n and nYF do
7          if a[i, j] = 0 then
8            nYF := false
9          else
10           j := j + 1;
11       if nYF then
12         i := i + 1
13     end
```

Fig. 6. The without-**goto** program for searching a 0-element in a two-dimensional array.

A solution with the **goto** statement is given in Fig. 5. Fig. 6 shows the program after eliminating the **goto** by introducing a Boolean variable nYF. Their proof-outlines are shown in Figs. 7 and 8, respectively.

The intuitive interpretation of predicate symbols and propositional constant symbols used in those proof-outlines are as follows: $NYF(k, l)$ means that no 0-element has been found yet (i.e., *N*ot *Y*et *F*ound); $MI(k, l)$ means that the pair of indices (k, l) is the lexicographically minimum one satisfying $a[k, l] = 0$ (i.e., *M*inimal *I*ndices); SZ means that the array a has at least one 0-element (i.e., *S*omewhere *Z*ero). These symbols satisfy the axioms in Fig. 9.

There are applications of the [Consequence] rule in proof-outlines in Figs. 7 and 8. For each application of [Consequence], we must show the validity of two implicational

```
1        {⊤}
2          {1 = 1}
3          i := 1
4          {i = 1}
5        {i = 1};
6          {NYF(i − 1, n)}
7          while i ≤ m do
8            {NYF(i − 1, n) ∧ i ≤ m}
9            begin
10             {NYF(i − 1, n) ∧ i ≤ m}
11               {NYF(i, 1 − 1) ∧ i ≤ m}
12               j := 1
13               {NYF(i, j − 1) ∧ i ≤ m}
14             {NYF(i, j − 1) ∧ i ≤ m};
15             while j ≤ n do
16               {NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n}
17               if a[i, j] = 0 then
18                 {NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ a[i, j] = 0}
19                   {MI(i, j) ∨ ¬SZ ∧ i > m}
20                   goto 99
21                   {⊥}
22                 {NYF(i, j − 1) ∧ i ≤ m}
23               else
24                 {NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ ¬(a[i, j] = 0)}
25                   {NYF(i, (j + 1) − 1) ∧ i ≤ m}
26                   j := j + 1
27                   {NYF(i, j − 1) ∧ i ≤ m}
28                 {NYF(i, j − 1) ∧ i ≤ m}
29               {NYF(i, j − 1) ∧ i ≤ m}
30             {NYF(i, j − 1) ∧ i ≤ m ∧ ¬(j ≤ n)};
31               {NYF((i + 1) − 1, n)}
32               i := i + 1
33               {NYF(i − 1, n)}
34             {NYF(i − 1, n)}
35           end
36           {NYF(i − 1, n)}
37         {NYF(i − 1, n) ∧ ¬(i ≤ m)}
38       {MI(i, j) ∨ ¬SZ ∧ i > m};
39   99: (∗ null statement ∗)
40       {MI(i, j) ∨ ¬SZ ∧ i > m}
```

Fig. 7. The complete proof-outline of the program in Fig. 5.

formulae (‘$A' \supset A$’ and ‘$B \supset B'$’ of this rule in Fig. 1). Showing the validity of each of these two formulae is a proof obligation of an application of the [Consequence] rule.

```
1   {⊤}
2     {1 = 1}
3     i := 1
4     {i = 1}
5   {i = 1};
6     {true ∧ i = 1}
7     nYF := true
8     {nYF ∧ i = 1}
9   {nYF ∧ i = 1};
10    {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
11    while i ≤ m and nYF do
12      {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)) ∧ i ≤ m and nYF}
13        {nYF ∧ NYF(i − 1, n) ∧ i ≤ m}
14        begin
15          {nYF ∧ NYF(i − 1, n) ∧ i ≤ m}
16            {nYF ∧ NYF(i, 1 − 1) ∧ i ≤ m}
17            j := 1
18            {nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
19          {nYF ∧ NYF(i, j − 1) ∧ i ≤ m};
20            {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
21            while j ≤ n and nYF do
22              {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m) ∧ j ≤ n and nYF}
23                {nYF ∧ NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n}
24                if a[i, j] = 0 then
25                  {nYF ∧ NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ a[i, j] = 0}
26                    {¬false ∧ MI(i, j)}
27                    nYF := false
28                    {¬nYF ∧ MI(i, j)}
29                  {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
30                else
31                  {nYF ∧ NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ ¬(a[i, j] = 0)}
32                    {nYF ∧ NYF(i, (j + 1) − 1) ∧ i ≤ m}
33                    j := j + 1
34                    {nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
35                  {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
36                {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
37              {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
38            {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, n) ∧ ¬(j ≤ n and nYF)}
39          {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, n)};
40          if nYF then
41            {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, n)) ∧ nYF}
42              {nYF ∧ NYF((i + 1) − 1, n)}
43              i := i + 1
44              {nYF ∧ NYF(i − 1, n)}
45            {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
46          {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
47        end
48        {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
49      {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
50    {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)) ∧ ¬(i ≤ m and nYF)}
51  {MI(i, j) ∨ ¬SZ ∧ i > m}
```

Fig. 8. The complete proof-outline of the program in Fig. 6.

$$
\begin{aligned}
MI(i,j) &\iff NYF(i,j-1) \wedge \mathrm{a}[i,j] = 0 \wedge 1{\leq}i{\leq}\mathrm{m} \wedge 1{\leq}j{\leq}\mathrm{n},\\
NYF(i,j) &\iff NYF(i,j-1) \wedge \mathrm{a}[i,j] \neq 0,\\
NYF(i,0) &\iff NYF(i-1,\mathrm{n}),\\
\neg SZ &\iff NYF(\mathrm{m},\mathrm{n}).
\end{aligned}
$$

Fig. 9. Axioms for predicates used in Figs. 7 and 8.

The axioms in Fig. 9 give sufficient properties to those symbols for proving proof obligations of proof-outlines in Figs. 7 and 8. Hence we do not give any concrete definition in a logical formula to each of those symbols.

Compared with the proof-outline in Fig. 7, the one in Fig. 8 looks much more complicated. There are two reasons why those two proof-outlines appear so different from each other.

(1) The loop invariant for the outer **while** loop is different: in Fig. 7 (with **goto**), it is a simple formula $NYF(i-1,n)$; in Fig. 8 (without **goto**), it is $\neg\mathrm{nYF} \wedge MI(\mathrm{i},\mathrm{j}) \vee \mathrm{nYF} \wedge NYF(\mathrm{i}-1,\mathrm{n})$.

(2) The number of applications of the [Consequence] rule is different: it is 6 in Fig. 7 while it is 10 in Fig. 8.

(1) causes (2); hence (1) is the essential point in the difference in appearance between those two proof-outlines. We therefore must analyze point (1) more carefully.

The loop invariant for the outer loop in the without-**goto** program is the disjunction of the following two formulae:

(a) the loop invariant for the corresponding loop in the with-**goto** program, $NYF(i-1,n)$, which holds in the case of no 0-element having been found yet, and

(b) $MI(i,j)$ which holds in the case of a 0-element having been found

after tagged (i.e., conjuncted) with $\neg$nYF and nYF (namely, the value of the Boolean variable nYF expressing which case holds), respectively.

This complication of the loop invariant for the outer loop in the without-**goto** program causes the assertion for every statement (e.g., the loop invariant for the inner loop) to be more complicated than that for the corresponding statement in the with-**goto** program.

Intuitively speaking, in proving the without-**goto** program in Fig. 6, we must take care of both cases, the 0-element-found case and the not-yet-found one, simultaneously for almost all statements inside the outer loop. On the other hand, in asserting the corresponding statements of the with-**goto** one in Fig. 5, we can forget the 0-element-found case except at the **goto** statement. In other words, the with-**goto** program allows us to separate two cases in asserting statements while we must be aware of both cases all the time in asserting the without-**goto** one. This is the source of the difference between the two proof-outlines in Figs. 7 and 8.

```
1   while whCond_1 do
2     begin
3       S_1;
4       ...
5         while whCond_{n+1} do
6           begin
7             S_{n+1};
8             if ifCond then
9               goto L
10            else
11              T_{n+1}
12          end
13      ...;
14      T_1
15    end;
16  U;
17  L: (* null statement *)
```

Fig. 10. The program-scheme of nested loops escaped with a **goto**.

3.2. Program schemes

We now generalize the above example-based discussion on the **goto** issue in escaping from nested loops to the level of program schemes. Before going into the discussion, we define a term: a *fresh* Boolean variable (nYF in the above example) introduced to eliminate **goto** is called a *'flag variable'*.

Fig. 10 is a program scheme with $(n+1)$ nested loops escaped by a **goto** statement from the innermost level. Note that each i-th level loop has the following form ($1 \leq i \leq n$):

```
while whCond_i do
  begin
    S_i;
    the (i + 1)-th level loop;
    T_i
  end
```

Fig. 11 is the corresponding scheme without any **goto** statement where fv is the flag variable introduced to eliminate the **goto**. In this case, each i-th level loop has the following form ($1 \leq i \leq n$):

```
while whCond_i and fv do
  begin
    S_i;
    the (i + 1)-th level loop;
    if fv then
      T_i
  end
```

For metavariables, each of S_i, T_i ($1 \leq i \leq n+1$) and U denotes a sequential composition of zero or more statements; each of $whCond_i$ ($1 \leq i \leq n+1$) and *ifCond* runs over the set of Boolean expressions; fv denotes a Boolean variable (flag variable).

```
 1    fv := true;
 2    while whCond_1 and fv do
 3      begin
 4        S_1;
 5        ...
 6          while whCond_{n+1} and fv do
 7            begin
 8              S_{n+1};
 9              if ifCond then
10                fv := false
11              else
12                T_{n+1}
13            end
14        ...;
15        if fv then
16          T_1
17      end;
18   if fv then
19     U
```

Fig. 11. The program-scheme after eliminating the **goto** in Fig. 10.

Note that, for the sake of simplicity, we only discuss the case with one **goto** statement for escaping from nested loops. It is not difficult, however, to generalize the result of this section to deal with any number of escaping **goto** statements.

Suppose a program X as an instance of the with-**goto** program scheme in Fig. 10. Let *Pre* and *Post* be its precondition and postcondition, respectively. Then the proof-outline of X is an instance of the proof-outline scheme in Fig. 12. Whenever the program X is correct with respect to these pre-/postconditions, there must be an intermediate assertion P and loop invariants Inv_i (for $1 \leq i \leq n+1$), for which every Hoare triple in the proof-outline in Fig. 12 is provable.

In that proof-outline, there are $(2n + 5)$ proof obligations which are nontrivial (i.e., cannot be shown by simple logical calculation). They can be classified into two categories. Proof obligations in the first category are on the validity of implicational formulae (for the correctness of applications of the [Consequence] rule) and they are the validity of ("#n" means the n-th line of the proof-outline):

(1) $Pre \supset Inv_1$,
i.e., the implication from #1 to #2;

(2) $P \wedge ifCond \supset Post$,
i.e., the implication from #18 to #19.

Proof obligations in the second category are on the provability of Hoare triples and they are to prove:

(3_i) $\{Inv_i \wedge whCond_i\}\, S_i\, \{Inv_{i+1}\}$ $(1 \leq i \leq n)$,
e.g., the Hoare triple in #6–8 $(i = 1)$;

(4_i) $\{Inv_{i+1} \wedge \neg whCond_{i+1}\}\, T_i\, \{Inv_i\}$ $(1 \leq i \leq n)$,
e.g., the Hoare triple in #32–34 $(i = 1)$;

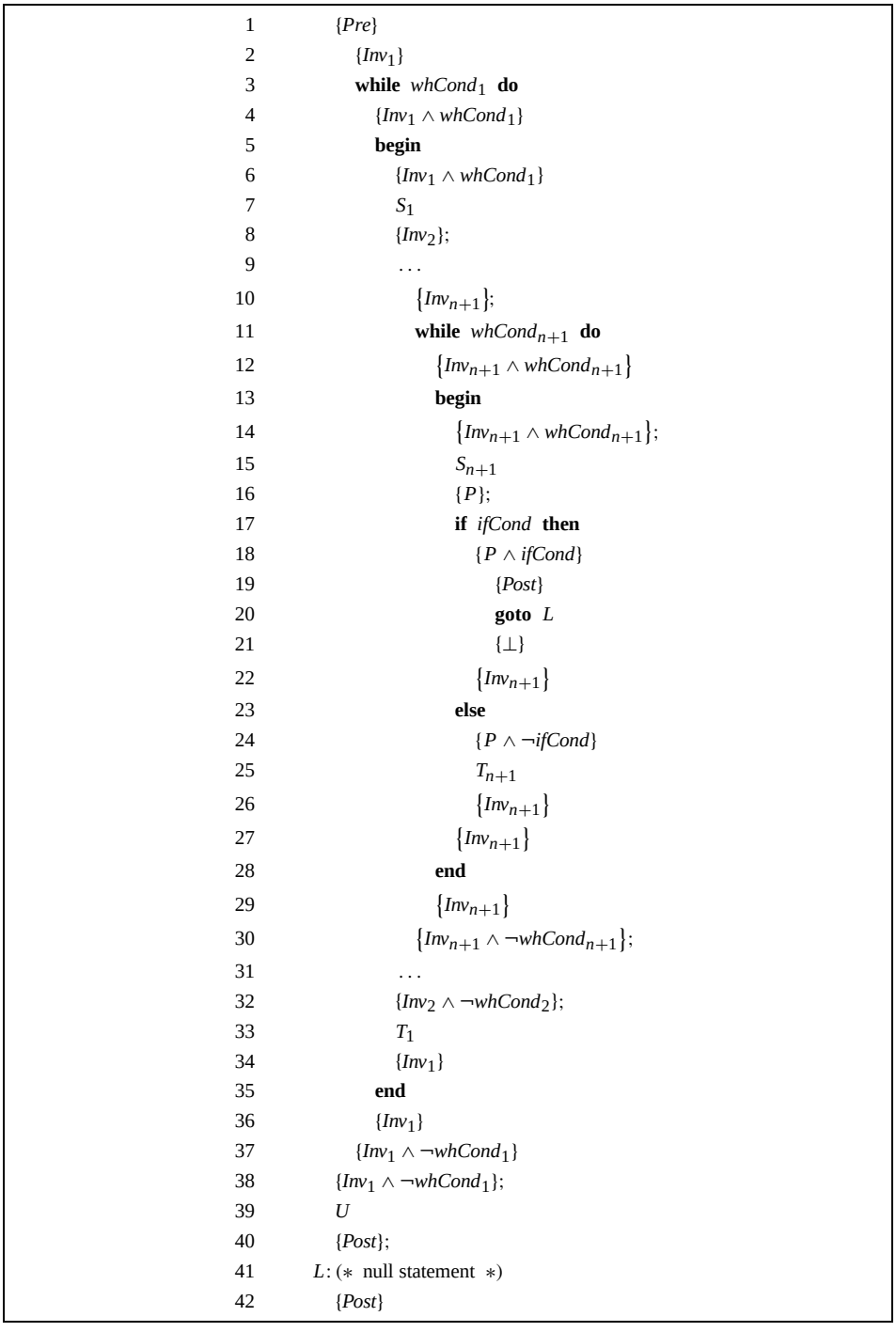


Fig. 12. The proof-outline for the program-scheme in Fig. 10.

(5) $\{Inv_{n+1} \wedge whCond_{n+1}\}\ S_{n+1}\ \{P\}$,
i.e., the Hoare triple in #14–16;
(6) $\{P \wedge \neg ifCond\}\ T_{n+1}\ \{Inv_{n+1}\}$,
i.e., the Hoare triple in #24–26;
(7) $\{Inv_1 \wedge \neg whCond_1\}\ U\ \{Post\}$,
i.e., the Hoare triple in #38–40.

Now suppose the corresponding **goto**-eliminated program Y as an instance of the without-**goto** program scheme in Fig. 11. Its proof-outline is an instance of the one shown in Fig. 13. If the program Y is correct with respect to *Pre* and *Post*, all proof obligations in Fig. 13 must be satisfied. They are:

(1′) $Pre \supset (\neg\text{true} \wedge Post \vee \text{true} \wedge Inv_1)$,
i.e., the implication from #1 to #2;
(2′) $(fv \wedge P \wedge ifCond) \supset (\neg\text{false} \wedge Post)$,
i.e., the implication from #27 to #28.
($3'_i$) $\{fv \wedge Inv_i \wedge whCond_i\}\ S_i\ \{fv \wedge Inv_{i+1}\}\ (1 \leq i \leq n)$,
e.g., the Hoare triple in #12–14 ($i = 1$);
($4'_i$) $\{fv \wedge Inv_{i+1} \wedge \neg whCond_{i+1}\}\ T_i\ \{fv \wedge Inv_i\}\ (1 \leq i \leq n)$,
e.g., the Hoare triple in #48–50 ($i = 1$);
(5′) $\{fv \wedge Inv_{n+1} \wedge whCond_{n+1}\}\ S_{n+1}\ \{fv \wedge P\}$,
i.e., the Hoare triple in #23–25;
(6′) $\{fv \wedge P \wedge \neg ifCond\}\ T_{n+1}\ \{fv \wedge Inv_{n+1}\}$,
i.e., the Hoare triple in #34–36;
(7′) $\{fv \wedge Inv_1 \wedge \neg whCond_1\}\ U\ \{fv \wedge Post\}$,
i.e., the Hoare triple in #60–62.

Since the flag variable fv is chosen as a fresh variable, it does not occur in S_i, T_i ($1 \leq i \leq n+1$), U, $whCond_i$ ($1 \leq i \leq n+1$) or *ifCond*. We can therefore safely assume that fv does not appear in *Pre*, *Post*, Inv_i ($1 \leq i \leq n+1$) or P.

Each of proof obligation (1′) $\sim$ (7′) for Fig. 13 can easily be derived from the corresponding one, (1) $\sim$ (7), for Fig. 12 as follows. As regards proof obligations on logical formulae, it is clear that (1) $\Longleftrightarrow$ (1′) and (2) $\Longrightarrow$ (2′). On derivations of (k') from (k) ($k = 3 \sim 7$), we note that the inference rule

$$\frac{\{A\}\ S\ \{C\} \quad \{B\}\ S\ \{D\}}{\{A \wedge B\}\ S\ \{C \wedge D\}},$$

and the axiom (where A does not contain any variables which can be assigned a value in S)

$$\{A\}\ S\ \{A\}$$

are admissible [1]. With these, it is easily shown that (k) $\Longrightarrow$ (k') for each $k = 3 \sim 7$.

The above paragraph means that proof obligations (1′) $\sim$ (7′) for the without-**goto** program scheme in Fig. 11 are logically weaker than those, (1) $\sim$ (7), for the with-**goto** program scheme in Fig. 10. In general, proving a logically weaker formula/triple is easier than proving a stronger one.

The above difference on the logical strength between (k) and (k') ($k = 1 \sim 7$), however, is merely superficial. This is simply due to our discussion being done on program schemes

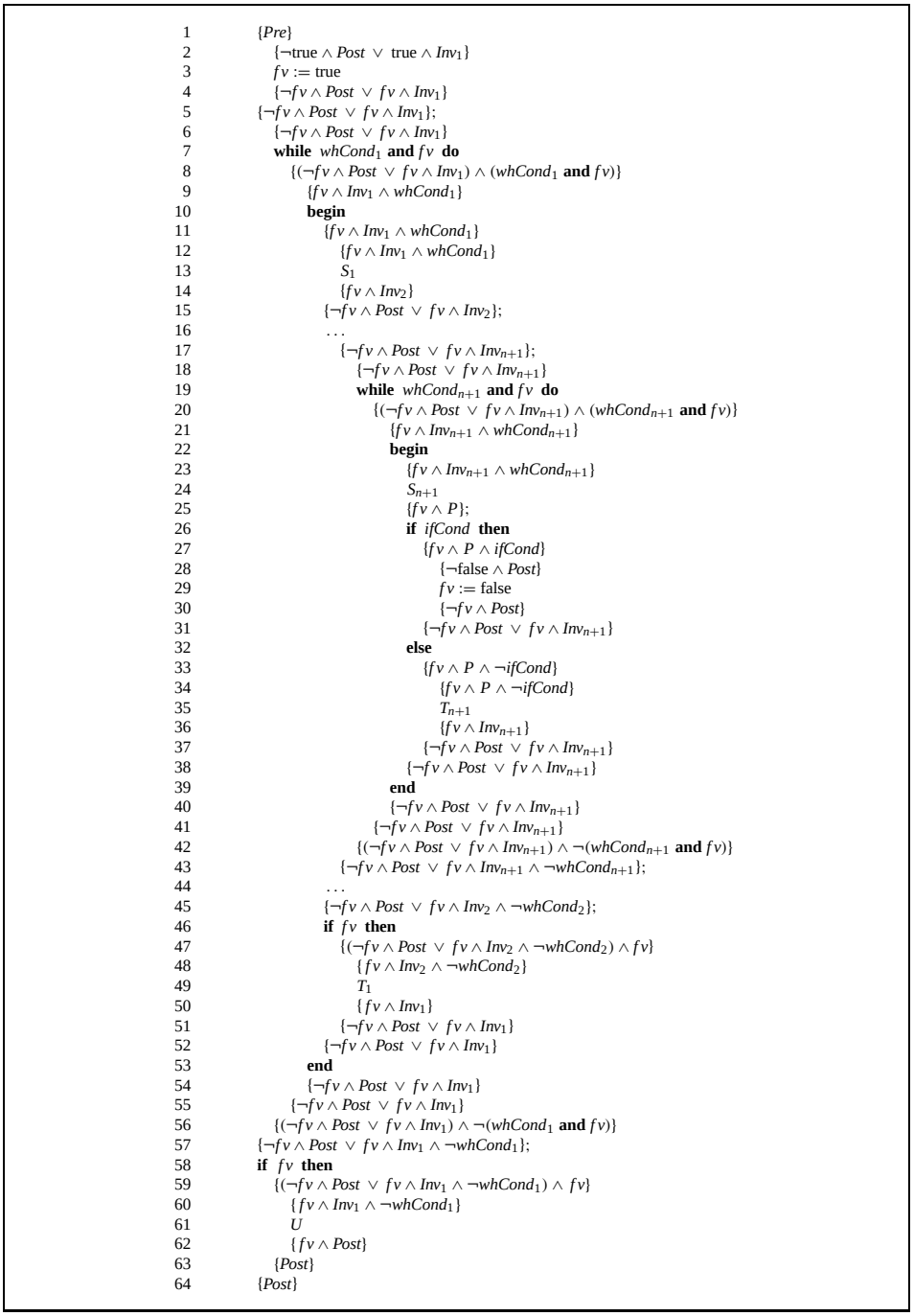


Fig. 13. The proof-outline for the **goto**-less program-scheme in Fig. 11.

containing metavariables. In concrete programs (like those in Figs. 5 and 6), their proof obligations have the same logical strength.

In the rest of this section, we show the last point. On (2) vs. (2′), note that fv does not occur in P, *Post* or *ifCond* by the choice of the flag variable fv. Now, if (2′) is valid, then (2′)[true/fv] must be true, too. The last formula is (true $\wedge$ P $\wedge$ *ifCond*) $\supset$ ($\neg$false $\wedge$ *Post*) and this is clearly equivalent to (2). Hence (2′) $\Longrightarrow$ (2) is shown, and with (2) $\Longrightarrow$ (2′) pointed out before, we get (2) $\Longleftrightarrow$ (2′).

On the equivalence between (k) and (k') ($k = 3 \sim 7$), the following simple lemma is enough.

Lemma 2. *Let S be a statement, A and B be formulae, and v be a* Boolean *variable not occurring in S, A or B. Then,*

$$\vdash \{v \wedge A\}\, S\, \{v \wedge B\} \implies \vdash \{A\}\, S\, \{B\}.$$

Proof. Since $\{v \wedge A\}\, S\, \{v \wedge B\}$ is provable, it is also provable when $v =$ true. That is, $\{(v \wedge A)[\text{true}/v]\}\, S[\text{true}/v]\, \{(v \wedge B)[\text{true}/v]\}$ is provable, too. Since v does not occur in S, A or B, $S[\text{true}/v] \equiv S$, $(v \wedge A)[\text{true}/v] \equiv v[\text{true}/v] \wedge A[\text{true}/v] \equiv \text{true} \wedge A \Longleftrightarrow A$ and $(v \wedge B)[\text{true}/v] \equiv \text{true} \wedge B \Longleftrightarrow B$. By the [Consequence] rule, $\{A\}\, S\, \{B\}$ is provable. □

Hence proof obligations (1) $\sim$ (7) for the with-**goto** program scheme are logically equivalent to those, (1′) $\sim$ (7′), for the without-**goto** one.

Moreover, just like our observation on the concrete example shown in the last subsection, the number of applications of the [Consequence] rule is larger and the loop invariant for the outermost **while** loop in the proof-outline of the without-**goto** program scheme is more complicated than that in the proof-outline (Fig. 12) of the with-**goto** one.

Therefore, it is fair to say that, for deep exit from nested loops, *using the* **goto** *statement makes the correctness proof easier.*

4. Case 2: Programming based on finite state machine modeling

In this section, the **goto** issue in the case of programming from a finite state machine model is considered. We first consider the example after Plauger. Then, like in the last section, we generalize the result to program schemes.

4.1. Example: Removing comments

We consider the following example originally given in Plauger's essay [52, Essay 4].

Example 3. Read a text from the input file ifl and write it to the output file ofl after removing every comment which starts with the begin-comment string `"/*"` and terminates with the end-comment string `"*/"`. Comments are not allowed to be nested; that is, the string `"/*"` within a comment is regarded as a part of that comment. When the last comment (if any) is incomplete (i.e., the end of file is encountered before the end-comment string is found), then process that incomplete comment as if it is complete.

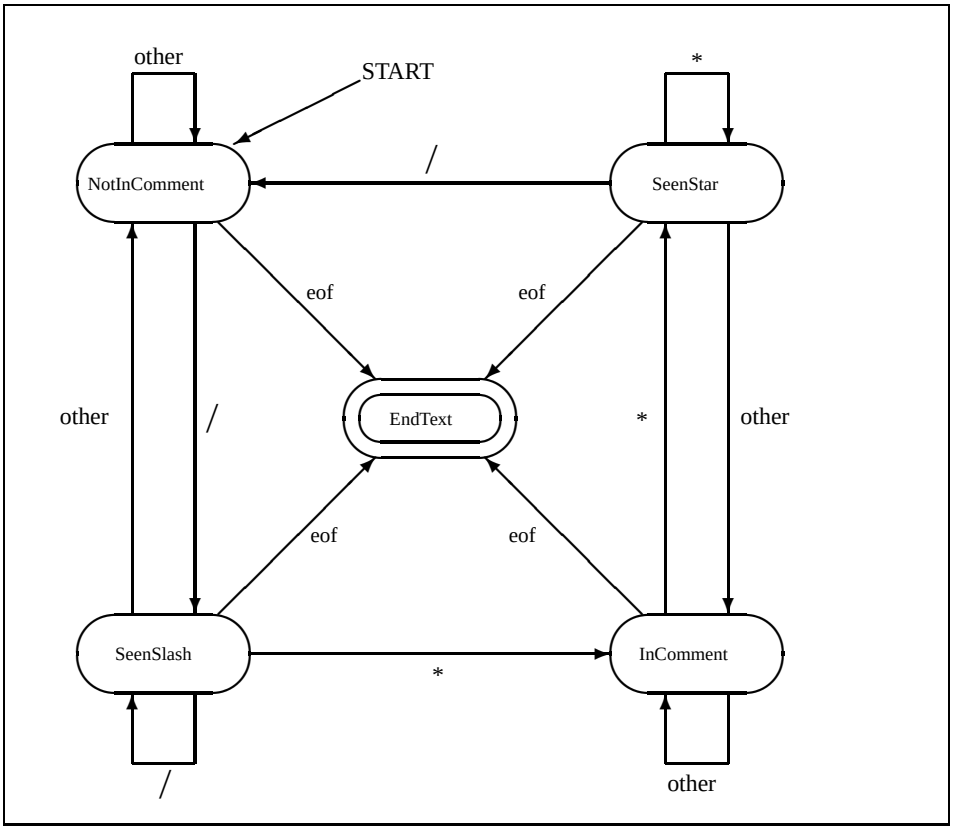


Fig. 14. The state transition diagram for removing comments.

Fig. 14 is the state transition diagram as the result of modeling this problem as a finite state machine.[2]

A program using a **goto** statement to express each state transition is shown in Fig. 15. Fig. 16 gives a without-**goto** one by introducing a fresh variable st whose type is an enumeration type (with enumeration constants representing 'states': NotInComment, SeenSlash, InComment and SeenStar), a **while** loop, and a **case** statement. We call such a newly introduced variable for recording the current state a *state variable*.

Note that, in order to minimize the difference in appearance between two programming styles due to syntactic restrictions of Pascal, the with-**goto** program in Fig. 15 has one deviation from the standard syntax of Pascal and also has two superfluous constructs: namely,

[2] This diagram and the program codes in Figs. 15 and 16 are essentially those given in Plauger's essay after making a few corrections in order to fix flaws in Plauger's original ones (we also translated the programs into Pascal). The flaws in his diagram and programs, however, are insignificant. What is important is his idea: namely, to use the **goto** statement is more suitable than not to use them in such a programming situation.

```
1   goto NotInComment;
2  NotInComment:
3   begin
4     get(ifl, c);
5     if c = eof then
6       goto EndText
7     else if c = '/' then
8       goto SeenSlash
9     else
10      begin
11        put(ofl, c);
12        goto NotInComment
13      end
14  end;
15 SeenSlash:
16  begin
17    get(ifl, c);
18    if c = eof then
19      begin
20        put(ofl, '/');
21        goto EndText
22      end
23    else if c = '/' then
24      begin
25        put(ofl, '/');
26        goto SeenSlash
27      end
28    else if c = '*' then
29      goto InComment
30    else
31      begin
32        put(ofl, '/');
33        put(ofl, c);
34        goto NotInComment
35      end
36  end;
37 InComment:
38  begin
39    get(ifl, c);
40    if c = eof then
41      goto EndText
42    else if c = '*' then
43      goto SeenStar
44    else
45      goto InComment
46  end;
47 SeenStar:
48  begin
49    get(ifl, c);
50    if c = eof then
51      goto EndText
52    else if c = '*' then
53      goto SeenStar
54    else if c = '/' then
55      goto NotInComment
56    else
57      goto InComment
58  end;
59 EndText:
60  (* null statement *)
```

Fig. 15. A program for removing comments (with **goto**).

```
1   st := NotInComment;
2   while st <> EndText do
3     case st of
4       NotInComment:
5         begin
6           get(ifl, c);
7           if c = eof then
8             st := EndText
9           else if c = '/' then
10            st := SeenSlash
11          else
12            begin
13              put(ofl, c);
14              st := NotInComment
15            end
16        end;
17      SeenSlash:
18        begin
19          get(ifl, c);
20          if c = eof then
21            begin
22              put(ofl, '/');
23              st := EndText
24            end
25          else if c = '/' then
26            begin
27              put(ofl, '/');
28              st := SeenSlash
29            end
30          else if c = '*' then
31            st := InComment
32          else
33            begin
34              put(ofl, '/');
35              put(ofl, c);
36              st := NotInComment
37            end
38        end;
39      InComment:
40        begin
41          get(ifl, c);
42          if c = eof then
43            st := EndText
44          else if c = '*' then
45            st := SeenStar
46          else
47            st := InComment
48        end;
49      SeenStar:
50        begin
51          get(ifl, c);
52          if c = eof then
53            st := EndText
54          else if c = '*' then
55            st := SeenStar
56          else if c = '/' then
57            st := NotInComment
58          else
59            st := InComment
60        end
61    end
```

Fig. 16. A program for removing comments (without **goto**).

(a) an identifier is used as the statement label representing each state because, in the without-**goto** program in Fig. 16, each state is expressed by an enumeration literal which is an identifier;
(b) for each state, a superfluous pair of **begin** and **end** is used to group statements of that state into single statement because each **case** branch of a **case** statement (used in the without-**goto** program) in Pascal must be single statement;
(c) a superfluous **goto** statement, "**goto** NotInComment", jumping to the initial state is added because the program in Fig. 16 needs to explicitly specify the initial by the assignment statement "st := NotInComment".

After these 'corrections' to the with-**goto** program, the remaining differences between two programs are as follows:

(1) each **goto** statement representing a state transition in Fig. 15 is replaced by an assignment statement to the *state variable* st in Fig. 16;
(2) the fact that repetitive state transitions are necessary to reach the final state is implicit in the with-**goto** program while it is explicitly expressed with a **while** loop in the without-**goto** program;
(3) in the with-**goto** program, the processing part for each state is automatically selected by jumping to the label at the first statement by a **goto** statement while an explicit selection by the **case** statement is necessary in the without-**goto** program.

Plauger claimed that these differences are insufficient to rationalize the introduction of a **while** loop and a **case** selection. He also argued that the with-**goto** program is more readable than the without-**goto** one.

Readability (or understandability) is rather a subjective measure, so we now compare these two programs from the viewpoint of Hoare Logic by analyzing their correctness proofs.

The precondition, *Pre*, of the removing comment problem is:

$$\mathrm{ifl}_{\mathrm{L}} = \varepsilon \wedge \mathrm{ofl}_{\mathrm{L}} = \varepsilon \wedge \mathrm{ifl}_{\mathrm{L}} :: \mathrm{ifl}_{\mathrm{R}} = \mathit{whole},$$

which expresses that nothing has been read from the input file ifl yet and nothing has been written to the output file ofl when the program is to start. The specification variable '*whole*' is introduced to keep the initial (and whole) contents of the input file.

The postcondition, *Post*, of this problem is:

$$\mathrm{ifl}_{\mathrm{L}} = \mathit{whole} \wedge \mathrm{ifl}_{\mathrm{R}} = \varepsilon \wedge \mathit{CommentRemoved}(\mathrm{ofl}_{\mathrm{L}}, \mathit{whole}),$$

which means that the input file has been completely read already and what has been written (i.e., $\mathrm{ofl}_{\mathrm{L}}$) is the text in which all comments are removed from *whole*.

In order to prove the correctness of a program modeled as a finite state machine, *it is clearly imperative that the precondition of each state (i.e., under what condition the state can be entered) must be clarified.*

In the case of the with-**goto** program in Fig. 15, this clarification of the precondition of each state just corresponds to the clarification of the precondition of each label representing that state. This exactly fits the moral on the use of the **goto** as we pointed out in Section 2.

$$
\begin{aligned}
\mathit{Pre} &\stackrel{\text{def}}{=} \mathrm{ifl}_\mathrm{L} = \varepsilon \wedge \mathrm{ofl}_\mathrm{L} = \varepsilon \wedge \mathit{FileInv},\\
\mathit{Post} &\stackrel{\text{def}}{=} \mathrm{ifl}_\mathrm{L} = \mathit{whole} \wedge \mathrm{ifl}_\mathrm{R} = \varepsilon\\
&\qquad\qquad \wedge \mathit{CommentRemoved}(\mathrm{ofl}_\mathrm{L}, \mathit{whole});\\
\\
\mathit{CommentRemoved}(t, s) &\stackrel{\text{def}}{=} \mathit{Removed}(t, s, \varepsilon);\\
\\
\mathit{PreNotInComment} &\stackrel{\text{def}}{=} \mathit{NotInCommentAux}(\mathrm{ofl}_\mathrm{L}, \mathrm{ifl}_\mathrm{L}),\\
\mathit{NotInCommentAux}(t, s) &\stackrel{\text{def}}{=} \mathrm{ifl}_\mathrm{R} = \varepsilon \wedge \mathit{Removed}(t, s, \mathrm{ifl}_\mathrm{R}) \wedge \mathit{FileInv}\\
&\quad \vee \mathrm{ifl}_\mathrm{R} \neq \varepsilon \wedge \mathit{SlashOrNot}(t, s, \mathrm{bwd}(\mathrm{ifl}_\mathrm{R}), \mathrm{fst}(\mathrm{ifl}_\mathrm{R}))\\
&\qquad\qquad \wedge \mathit{FileInv}',\\
\mathit{SlashOrNot}(t, s, r, c) &\stackrel{\text{def}}{=} c = \texttt{'/'} \wedge \mathit{SlashAdded}(t \mathbin{:\cdot} c, s \mathbin{:\cdot} c, r)\\
&\quad \vee c \neq \texttt{'/'} \wedge \mathit{Removed}(t \mathbin{:\cdot} c, s \mathbin{:\cdot} c, r);\\
\\
\mathit{PreSeenSlash} &\stackrel{\text{def}}{=} \mathit{SlashAux}(\mathrm{ofl}_\mathrm{L}, \mathrm{ifl}_\mathrm{L}),\\
\mathit{SlashAux}(t, s) &\stackrel{\text{def}}{=} \mathrm{ifl}_\mathrm{R} = \varepsilon \wedge \mathit{SlashAdded}(t \mathbin{:\cdot} \texttt{'/'}, s, \mathrm{ifl}_\mathrm{R}) \wedge \mathit{FileInv}\\
&\quad \vee \quad \mathrm{ifl}_\mathrm{R} \neq \varepsilon\\
&\qquad \wedge \mathit{SlashStarOrOther}(t \mathbin{:\cdot} \texttt{'/'}, s, \mathrm{bwd}(\mathrm{ifl}_\mathrm{R}), \mathrm{fst}(\mathrm{ifl}_\mathrm{R}))\\
&\qquad\qquad \wedge \mathit{FileInv}',\\
\mathit{SlashStarOrOther}(t, s, r, c) &\stackrel{\text{def}}{=} c = \texttt{'/'} \wedge \mathit{SlashAdded}(t \mathbin{:\cdot} c, s \mathbin{:\cdot} c, r)\\
&\quad \vee c = \texttt{'*'} \wedge \mathit{Skipping}(\mathrm{fwd}(t), s \mathbin{:\cdot} c, r)\\
&\quad \vee c \notin \{\texttt{'/'}, \texttt{'*'}\} \wedge \mathit{Removed}(t \mathbin{:\cdot} c, s \mathbin{:\cdot} c, r);\\
\\
\mathit{PreInComment} &\stackrel{\text{def}}{=} \mathit{InCommentAux}(\mathrm{ofl}_\mathrm{L}, \mathrm{ifl}_\mathrm{L}),\\
\mathit{InCommentAux}(t, s) &\stackrel{\text{def}}{=} \mathrm{ifl}_\mathrm{R} = \varepsilon \wedge \mathit{Skipping}(t, s, \mathrm{ifl}_\mathrm{R}) \wedge \mathit{FileInv}\\
&\quad \vee \mathrm{ifl}_\mathrm{R} \neq \varepsilon \wedge \mathit{StarOrNot}(t, s, \mathrm{bwd}(\mathrm{ifl}_\mathrm{R}), \mathrm{fst}(\mathrm{ifl}_\mathrm{R}))\\
&\qquad\qquad \wedge \mathit{FileInv}',\\
\mathit{StarOrNot}(t, s, r, c) &\stackrel{\text{def}}{=} c = \texttt{'*'} \wedge \mathit{StarAdded}(t, s \mathbin{:\cdot} c, r)\\
&\quad \vee c \neq \texttt{'*'} \wedge \mathit{Skipping}(t, s \mathbin{:\cdot} c, r);\\
\\
\mathit{PreSeenStar} &\stackrel{\text{def}}{=} \mathit{StarAux}(\mathrm{ofl}_\mathrm{L}, \mathrm{ifl}_\mathrm{L}),\\
\mathit{StarAux}(t, s) &\stackrel{\text{def}}{=} \mathrm{ifl}_\mathrm{R} = \varepsilon \wedge \mathit{Removed}(t, s) \wedge \mathit{FileInv}\\
&\quad \vee \mathrm{ifl}_\mathrm{R} \neq \varepsilon \wedge \mathit{StarSlashOrOther}(t, s, \mathrm{bwd}(\mathrm{ifl}_\mathrm{R}), \mathrm{fst}(\mathrm{ifl}_\mathrm{R}))\\
&\qquad\qquad \wedge \mathit{FileInv}',\\
\mathit{StarSlashOrOther}(t, s, r, c) &\stackrel{\text{def}}{=} c = \texttt{'*'} \wedge \mathit{StarAdded}(t, s \mathbin{:\cdot} c, r)\\
&\quad \vee c = \texttt{'/'} \wedge \mathit{Removed}(t, s \mathbin{:\cdot} c, r)\\
&\quad \vee c \notin \{\texttt{'*'}, \texttt{'/'}\} \wedge \mathit{Skipping}(t, s \mathbin{:\cdot} c, r);\\
\\
\mathit{FileInv} &\stackrel{\text{def}}{=} \mathrm{ifl}_\mathrm{L} :: \mathrm{ifl}_\mathrm{R} = \mathit{whole},\\
\mathit{FileInv}' &\stackrel{\text{def}}{=} (\mathrm{ifl}_\mathrm{L} \mathbin{:\cdot} \mathrm{fst}(\mathrm{ifl}_\mathrm{R})) :: \mathrm{bwd}(\mathrm{ifl}_\mathrm{R}) = \mathit{whole}.
\end{aligned}
$$

Fig. 17. Pre-/postconditions of the program, preconditions of state and their auxiliary predicates.

Preconditions of all states and related auxiliary predicates are summarized in Fig. 17. The axioms satisfied by auxiliary predicates whose definition is not shown there are axiomatically defined in Fig. 18. Intuitively speaking, these axioms simulate state transitions at the level of logic. We therefore call those predicates '*state transition predicates*'. These axioms are enough to show proof obligations in proof-outlines of those two programs; hence we do not give a concrete definition with a logical formula to each state transition predicate.

Every state transition predicate takes three arguments. For example, $\mathit{Removed}(t, s, r)$ intuitively means that the *t*arget string t is the string in which all comments are *removed* from the *s*ource string s under the context of the *r*est string r. The reason why the third argument r is necessary is that, for example, it is not possible to judge whether the current character `'/'` is just the first character of the begin-comment string `"/*"` or not without consulting the rest string r as the context.

$$
\begin{aligned}
& \Longrightarrow \mathit{Removed}(\varepsilon, \varepsilon, r), \\
\mathit{Removed}(t, s, c \cdot\!: r) \wedge c \neq \texttt{'/'} & \Longrightarrow \mathit{Removed}(t :\!\cdot c, s :\!\cdot c, r), \\
\mathit{Removed}(t, s, \texttt{'/'} \cdot\!: r) & \Longrightarrow \mathit{SlashAdded}(t :\!\cdot \texttt{'/'}, s :\!\cdot \texttt{'/'}, r), \\
\\
\mathit{SlashAdded}(t :\!\cdot \texttt{'/'}, s :\!\cdot \texttt{'/'}, \varepsilon) & \Longrightarrow \mathit{Removed}(t :\!\cdot \texttt{'/'}, s :\!\cdot \texttt{'/'}, \varepsilon), \\
\mathit{SlashAdded}(t :\!\cdot \texttt{'/'}, s :\!\cdot \texttt{'/'}, c \cdot\!: r) \wedge c \neq \texttt{'/'} \wedge c \neq \texttt{'*'} & \Longrightarrow \mathit{Removed}(t :\!\cdot \texttt{'/'} :\!\cdot c, s :\!\cdot \texttt{'/'} :\!\cdot c, r), \\
\mathit{SlashAdded}(t :\!\cdot \texttt{'/'}, s :\!\cdot \texttt{'/'}, \texttt{'/'} \cdot\!: r) & \Longrightarrow \mathit{SlashAdded}(t :\!\cdot \texttt{'/'} :\!\cdot \texttt{'/'}, s :\!\cdot \texttt{'/'} :\!\cdot \texttt{'/'}, r), \\
\mathit{SlashAdded}(t :\!\cdot \texttt{'/'}, s :\!\cdot \texttt{'/'}, \texttt{'*'} \cdot\!: r) & \Longrightarrow \mathit{Skipping}(t, s :: \texttt{"/*"}, r), \\
\\
\mathit{Skipping}(t, s :: \texttt{"/*"} :: u, \varepsilon) & \Longrightarrow \mathit{Removed}(t, s :: \texttt{"/*"} :: u, \varepsilon), \\
\mathit{Skipping}(t, s :: \texttt{"/*"} :: u, c \cdot\!: r) \wedge c \neq \texttt{'*'} & \Longrightarrow \mathit{Skipping}(t, s :: \texttt{"/*"} :: u :\!\cdot c, r), \\
\mathit{Skipping}(t, s :: \texttt{"/*"} :: u, \texttt{'*'} \cdot\!: r) & \Longrightarrow \mathit{StarAdded}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'}, r), \\
\\
\mathit{StarAdded}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'}, \varepsilon) & \Longrightarrow \mathit{Removed}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'}, \varepsilon), \\
\mathit{StarAdded}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'}, c \cdot\!: r) \wedge c \neq \texttt{'*'} \wedge c \neq \texttt{'/'} & \Longrightarrow \mathit{Skipping}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'} :\!\cdot c, r), \\
\mathit{StarAdded}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'}, \texttt{'*'} \cdot\!: r) & \Longrightarrow \mathit{StarAdded}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'} :\!\cdot \texttt{'*'}, r), \\
\mathit{StarAdded}(t, s :: \texttt{"/*"} :: u :\!\cdot \texttt{'*'}, \texttt{'/'} \cdot\!: r) & \Longrightarrow \mathit{Removed}(t, s :: \texttt{"/*"} :: u :: \texttt{"*/"}, r).
\end{aligned}
$$

Fig. 18. Axioms for state transition predicates.

The part for the state NotInComment of the proof-outline for the with-**goto** program is shown in Fig. 19. The proof-outline for other states can be constructed similarly.

Note that, in this proof-outline, the phrase ‘the precondition of the textually following state’ is used in several assertions. This phrase is to express the fact that those assertions are locally insignificant, where ‘locally’ means that ‘considering the part for the state NotInComment alone’. According to the particular ordering of states in Fig. 15, the state NotInComment is followed by SeenSlash; hence this phrase means *PreSeenSlash*, the precondition of SeenSlash.

The above paragraph means the following important nature of the correctness proof of the with-**goto** program designed by a finite state machine model: *after clarifying the preconditions of states, we can prove the correctness of the with-***goto** *program state by state; i.e., the correctness of each state can be proved independently.*

On the other hand, in order to construct the proof-outline of the without-**goto** program, the loop invariant for the **while** loop must be found, but there is no way to determine this loop invariant in the top-down (outside-in) manner because there is only one assignment statement around the **while** loop and this statement cannot sufficiently restrict the precondition (i.e., the loop invariant) of the **while** loop.

We therefore must find the loop invariant in the bottom-up manner, i.e., from the precondition of the loop body. The loop body is a **case** statement; hence its precondition must be determined by preconditions of all **case**-branches, i.e., preconditions of all states. Thus, we obtain the loop invariant as follows:

1 {*PreNotInComment*}
2 NotInComment:
3 **begin**
4 {*PreNotInComment*}
5 get(ifl, c)
6 $\left\{ \begin{array}{l} \mathrm{c} = \mathrm{eof} \wedge \mathrm{ifl_R} = \varepsilon \wedge \mathit{Removed}(\mathrm{ofl_L}, \mathrm{ifl_L}, \mathrm{ifl_R}) \wedge \mathit{FileInv} \\ \vee\ \mathrm{c} \neq \mathrm{eof} \wedge \left(\begin{array}{l} \mathrm{c} = \texttt{'/'} \wedge \mathit{SlashAdded}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \\ \vee\ \mathrm{c} \neq \texttt{'/'} \wedge \mathit{Removed}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \end{array} \right) \\ \qquad \wedge \mathit{FileInv} \end{array} \right\};$
7 **if** c = eof **then**
8 $\left\{ \left(\begin{array}{l} \mathrm{c} = \mathrm{eof} \wedge \mathrm{ifl_R} = \varepsilon \wedge \mathit{Removed}(\mathrm{ofl_L}, \mathrm{ifl_L}, \mathrm{ifl_R}) \wedge \mathit{FileInv} \\ \vee\ \mathrm{c} \neq \mathrm{eof} \wedge \left(\begin{array}{l} \mathrm{c} = \texttt{'/'} \wedge \mathit{SlashAdded}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \\ \vee\ \mathrm{c} \neq \texttt{'/'} \wedge \mathit{Removed}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \end{array} \right) \\ \qquad \wedge \mathit{FileInv} \end{array} \right) \wedge \mathrm{c} = \mathrm{eof} \right\}$
9 {*Post*}
10 **goto** EndText
11 {⊥}
12 {the precondition of the textually following state}
13 **else if** c = '/' **then**
14 $\left\{ \left(\begin{array}{l} \mathrm{c} = \mathrm{eof} \wedge \mathrm{ifl_R} = \varepsilon \wedge \mathit{Removed}(\mathrm{ofl_L}, \mathrm{ifl_L}, \mathrm{ifl_R}) \wedge \mathit{FileInv} \\ \vee\ \mathrm{c} \neq \mathrm{eof} \wedge \left(\begin{array}{l} \mathrm{c} = \texttt{'/'} \wedge \mathit{SlashAdded}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \\ \vee\ \mathrm{c} \neq \texttt{'/'} \wedge \mathit{Removed}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \end{array} \right) \\ \qquad \wedge \mathit{FileInv} \end{array} \right) \wedge \neg(\mathrm{c} = \mathrm{eof}) \wedge \mathrm{c} = \texttt{'/'} \right\}$
15 {*PreSeenSlash*}
16 **goto** SeenSlash
17 {⊥}
18 {the precondition of the textually following state}
19 **else**
20 $\left\{ \left(\begin{array}{l} \mathrm{c} = \mathrm{eof} \wedge \mathrm{ifl_R} = \varepsilon \wedge \mathit{Removed}(\mathrm{ofl_L}, \mathrm{ifl_L}, \mathrm{ifl_R}) \wedge \mathit{FileInv} \\ \vee\ \mathrm{c} \neq \mathrm{eof} \wedge \left(\begin{array}{l} \mathrm{c} = \texttt{'/'} \wedge \mathit{SlashAdded}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \\ \vee\ \mathrm{c} \neq \texttt{'/'} \wedge \mathit{Removed}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L}, \mathrm{ifl_R}) \end{array} \right) \\ \qquad \wedge \mathit{FileInv} \end{array} \right) \wedge \neg(\mathrm{c} = \mathrm{eof}) \wedge \neg(\mathrm{c} = \texttt{'/'}) \right\}$
21 {$\mathrm{c} = \mathrm{eof} \wedge \mathit{NotInCommentAux}(\mathrm{ofl_L}, \mathrm{ifl_L}) \vee \mathrm{c} \neq \mathrm{eof} \wedge \mathit{NotInCommentAux}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L})$}
22 **begin**
23 {$\mathrm{c} = \mathrm{eof} \wedge \mathit{NotInCommentAux}(\mathrm{ofl_L}, \mathrm{ifl_L}) \vee \mathrm{c} \neq \mathrm{eof} \wedge \mathit{NotInCommentAux}(\mathrm{ofl_L} :\cdot \mathrm{c}, \mathrm{ifl_L})$}
24 put(ofl, c)
25 {*PreNotInComment*};
26 **goto** NotInComment
27 {⊥}
28 **end**
29 {⊥}
30 {the precondition of the textually following state}
31 {the precondition of the textually following state}
32 **end**
33 {the precondition of the textually following state}

Fig. 19. The proof-outline of the state NotInComment in Fig. 15.

$$
\begin{aligned}
&\mathrm{st} = \mathrm{EndText} \wedge \mathit{Post}\\
\vee\ &\mathrm{st} = \mathrm{NotInComment} \wedge \mathit{PreNotInComment}\\
\vee\ &\mathrm{st} = \mathrm{SeenSlash} \wedge \mathit{PreSeenSlash}\\
\vee\ &\mathrm{st} = \mathrm{InComment} \wedge \mathit{PreInComment}\\
\vee\ &\mathrm{st} = \mathrm{SeenStar} \wedge \mathit{PreSeenStar}
\end{aligned}
$$

This loop invariant intuitively means that *the current state is one of possible states, and the precondition of that state holds*. Hence this loop invariant carries hardly any useful information for the correctness proof.

With this loop invariant, the proof-outline of the NotInComment state in the without-**goto** program can be constructed as shown in Fig. 20.

The correspondence between this proof-outline and the one in Fig. 19 is clear, but the assertions in the former are more complicated than those in the latter due to components (of conjunction) for house-keeping the value of the state variable st.

As a summary, proving the with-**goto** program is simpler than proving the without-**goto** program. At least, assertions for the former are simpler than the corresponding ones for the latter. Even worse, there is no way (other than in the bottom-up manner) to find the loop invariant of the **while** loop, which was introduced *to structure* the program.

Therefore, Plauger's claim that state transitions should be expressed by the **goto** statement in programming from a finite state machine model and the with-**goto** program is more understandable than the without-**goto** one is scientifically and objectively supported by Hoare Logic.

4.2. Program schemes

We now generalize the above discussion on the **goto** issue for the class of programs by finite state machine modeling.

Fig. 21 shows both the with-**goto** program scheme and the without-**goto** one (with a state variable *sv*) of the class of programs which read data from the input file *infile* into the variable v. In those program schemes, Σ_1 is the initial state and Σ_{n+1} is the final state. For metavariables, Σ_i represents a state (i.e., a statement label in the with-**goto** scheme or an enumeration constant in the without-**goto** one); $cond_{i,j}$ is a Boolean expression; $F_{i,j}$ is a (possibly empty) sequential composition of statements.

The Σ_i-state part of the proof-outline for the with-**goto** program scheme is given in Fig. 22. That for the without-**goto** one is shown in Fig. 23. Note that the precondition of the final state, $Pre\Sigma_{n+1}$, is just the postcondition of the whole program, *Post*.

The correspondence between those two proof-outlines is clear, but, as we observed in Plauger's example, the loop invariant of the 'structured' (without-**goto**) one is the disjunction of every precondition (of each state), which is tagged (conjuncted) with an equation showing which state the precondition is for. Again the proof-outline of the 'structured' scheme is forced to be more complicated in order to maintain such state information.

Therefore, it can be said that, for programs designed with finite state machine modeling, *using the* **goto** *statement makes the correctness proof simpler.*

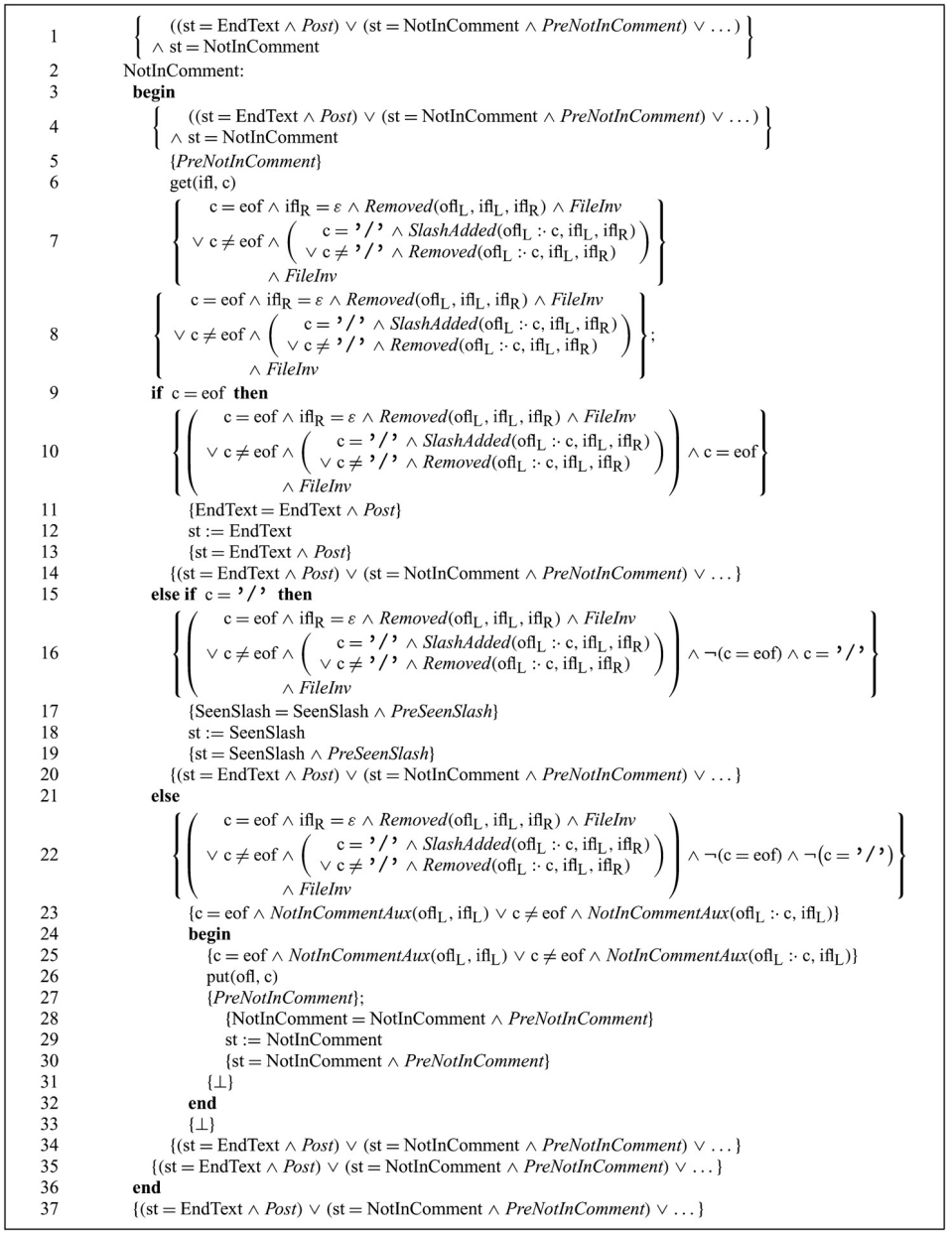


Fig. 20. The proof-outline of the state NotInComment in Fig. 16.

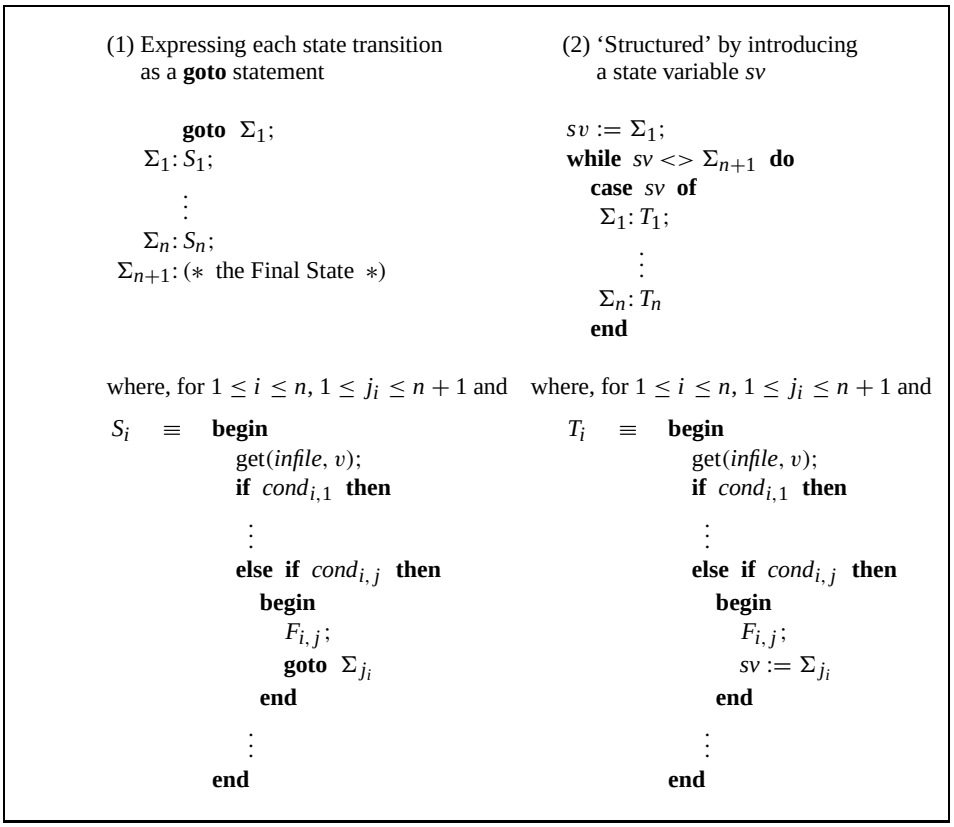


Fig. 21. Program schemes based on finite state machine modeling (both with- and without-**goto**).

5. Conclusion — On programming styles from the program verification viewpoint

First of all, the result of the present work should not be sloppily quoted as "the **goto** statement makes correctness proofs easier". In fact, the [Goto] rule in Fig. 1 has a more complicated form than other rules. The premises of this rule have to have the entailment symbol '$\vdash$' explicitly, and this fact means that this rule is technically more complicated than other rules, and conceptually the presence of this rule makes the system of Hoare Logic more difficult. Exactly speaking, in the presence of [Goto], Hoare triples as premises or the conclusion of all other axioms/rules must have a hypothesis part (a finite collection of Hoare triples). For example, the [Sequencing] rule in Fig. 1 should have been given as

$$\frac{\Gamma \vdash \{A\}\, S_1\, \{B\} \qquad \Gamma \vdash \{B\}\, S_2\, \{C\}}{\Gamma \vdash \{A\}\, S_1; S_2\, \{C\}}$$

and the [Goto] rule should have been shown as

$$\frac{\Gamma, \{B\}\,\mathbf{goto}\ L\ \{\bot\} \vdash \{A\}\, S_1\, \{B\} \qquad \Gamma, \{B\}\,\mathbf{goto}\ L\ \{\bot\} \vdash \{B\}\, S_2\, \{C\}}{\Gamma \vdash \{A\}\, S_1; L\!:\! S_2\, \{C\}}$$

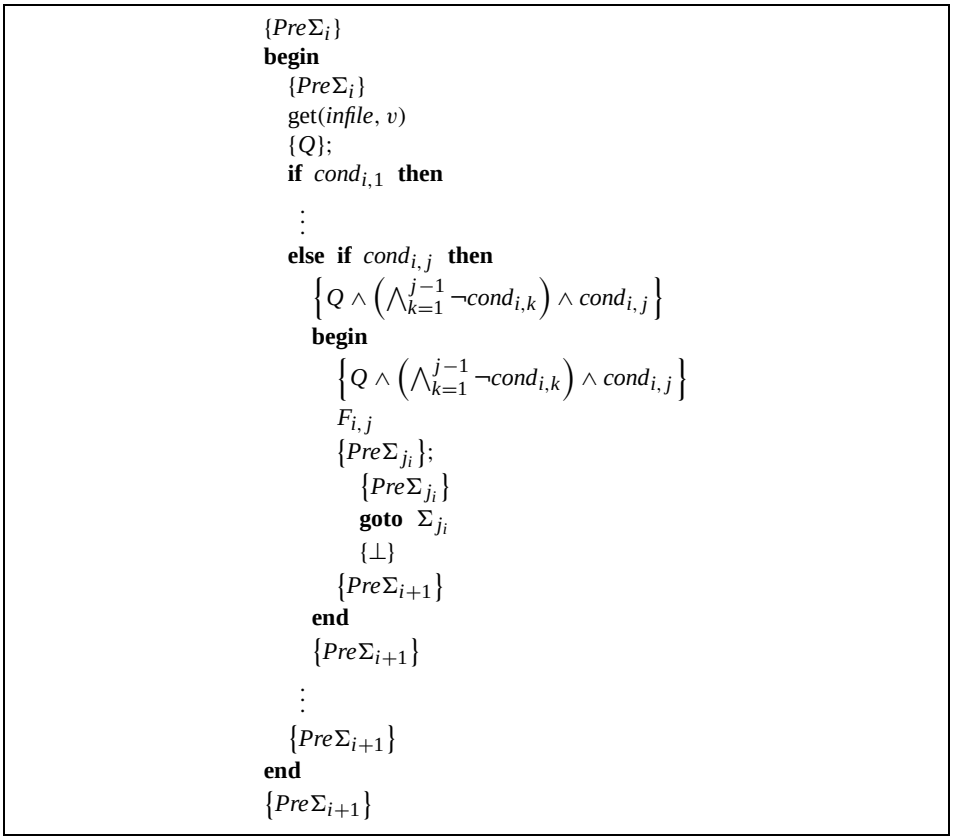


Fig. 22. The proof-outline of S_i for the state Σ_i in the with-**goto** program scheme.

where Γ is the hypothesis part and consists of a finite (possibly empty) collection of Hoare triples. All other rules must be rewritten just like [Sequencing]. We also need structural rules (weakening, exchange, contraction) for manipulating Hoare triples in the hypothesis part just like in Gentzen's sequent calculus. The reason why we presented the axioms and rules as shown in Fig. 1 is that it is a well-accepted convention to shown only non-trivial triples as hypotheses in the [Goto] rule and to omit other hypotheses. But omitting them is merely a convention and the necessity of them becomes clear when we are going to fully formalize our Hoare Logic system, say on a mechanical proof checker like Isabelle, HOL, Coq, LF, ... etc. In other words, when we admit the use of the **goto** statement and include the [Goto] rule into our Hoare Logic system, we must raise the level of thinking from the one on Hoare triples to the one on sequents (of Hoare triples).

Hence we should avoid such a complicated programming device as far as possible. What we have shown in the present work is that such a technical difficulty of the **goto** statement can, however, be paid in some programming situations by the simplification of assertions.

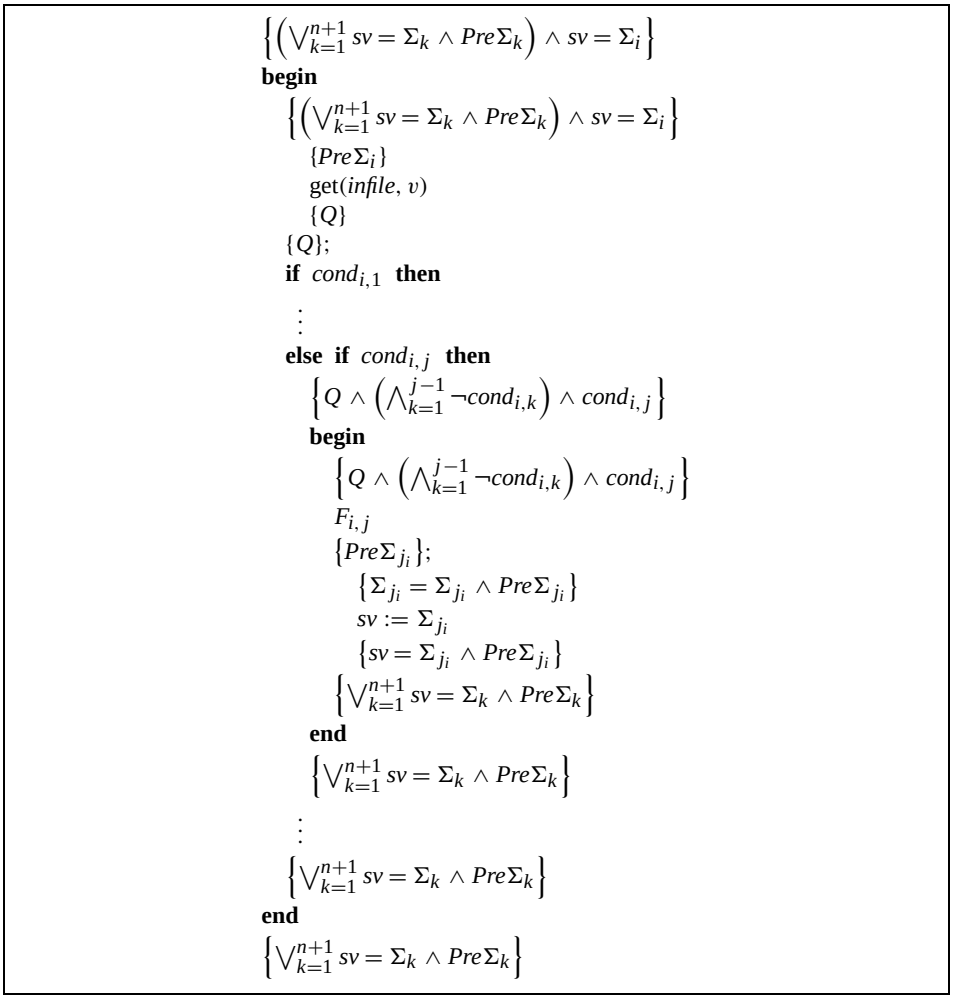


Fig. 23. The proof-outline of S_i for the state Σ_i in the without-**goto** program scheme.

We now return to the discussion on our results. In this work, we have shown that, in two cases, the with-**goto** styles are simpler than the directly corresponding without-**goto** ones from the viewpoint of the correctness proof in Hoare Logic. Cases which we have discussed are on deep exit from nested loops and on state transitions in programs designed through finite state machine modeling.

The common feature of both cases is that we have to introduce a new variable into the without-**goto** style in order to eliminate **goto** statements: a flag variable in the case of escaping from nested loops and a state variable in the case of state transitions.

This is the key why proof-outlines of the without-**goto** styles are more complicated than those of the corresponding with-**goto** ones. The value of a program variable can be changed dynamically; hence loop invariants of without-**goto** programs had to be of the form:

$$\bigvee_i (variable = value_i) \wedge condition_i$$

On the other hand, in the corresponding with-**goto** ones, the simple assertion, $condition_i$, is sufficient for each part where the value of *variable* is just $value_i$.

This phenomenon exactly reflects the very last sentences of [15] by Dijkstra: *The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.*

Mills and his colleagues strongly pushed the **goto**-less programming style on the basis of theoretical results on the expressibility of control flows. In particular, in their monograph [41, Section 4.4.3], they give a 'Structure Theorem'. Their proof of this theorem, however, is just based on a mechanical transformation by introducing an integer variable whose value represents *which part of the original (maybe with-***goto***) program is now executing*. In fact, their transformation is just the source-to-source translation technique used in the construction of a universal interpreter (e.g., [35, p. 59]) in computability theory.

Hence, when we prove the correctness of such a transformed program by Hoare Logic, it is clear that the complication of assertions, just like observed in the present work, occurs.

To the results of this work, there may be a critique like "In general, if we start with a program in one language $\mathcal{L}_\mathcal{A}$, say Pascal with **goto**, and then mechanically translate it to other language $\mathcal{L}_\mathcal{B}$, say Pascal without **goto**, then the resulting program becomes longer in $\mathcal{L}_\mathcal{B}$. In the cases in the present work, this mechanical translation introduces new variables with assignments to them and test on them. If we use a mechanical translation scheme inversely from $\mathcal{L}_\mathcal{B}$ to $\mathcal{L}_\mathcal{A}$, then we must remove a Boolean variable (in the case of deep exit) and the resulting program in $\mathcal{L}_\mathcal{A}$ becomes explosively longer than the original one in $\mathcal{L}_\mathcal{B}$. This means that the length of programs (and the length of assertions for them) depends on the direction of translation".

We agree with the last claim: i.e., the increase of the program length depends on the direction of translation. The above critique partially applies to the first case on the deep exit from nested loops. The without-**goto** program in Fig. 6 can be regarded as the resultant of a mechanical translation of the with-**goto** one in Fig. 5. With respect to the program schemes in Figs. 10 and 11, the same critique may hold. We now must take this critique very seriously and discuss how it applies to our results.

Let us consider this critique on our first case. In this case, our main intension is to show what the critique claims: i.e., the uselessness of such a mechanical translation from the correctness proof viewpoint. Why do we criticize this fact in the present work? It is because, although many programmers wisely use the with-**goto** style (Fig. 10) in this case, such a mechanical translation is still widely accepted in industrial disciplines: i.e., a nonnegligible number of real-world programmers still force themselves to write deep exits in the without-**goto** style in Fig. 11.

This critique does not apply at all to the second case on programming from finite state modeling. In this case, the translation should not be regarded as the one from the with-**goto** program in Fig. 15 to the without-**goto** one in Fig. 16. Instead, the actual translation is from the state transition diagram in Fig. 14 to each of those programs. In other words, the translation in this case is a selection of the representing method for state transitions:

either with **goto** statements or with a state variable and assignments. Hence, in this case, those two programs are in the equal position: i.e., either one is not the translation from the other. Our result shows that, from the correctness proof viewpoint, we should use **goto** statements to represent state transitions.

Then a question arises why Mills and his colleagues insisted upon such a mechanical elimination of **goto** statements despite Dijkstra's clear warning as quoted above. Needless to say, there might have been many reasons and some of them might be social or philosophical. But one of the possible technical reasons is that they adopted the functional equivalence on state transformations [41,47] as the criterion of their program verification.

As textbooks on denotational semantics (e.g., [57]) tell us, we can use the direct semantics for programming languages without jumps (nor exceptions); i.e., we can interpret a statement as a state transformer (endofunction on the set of states). When we introduce jumps into our language, we must use the continuation semantics in which a statement is interpreted as a continuation transformer. Since a continuation itself is a function, the denotation of a statement becomes a higher-order function, for which is too difficult for practitioners to have intuitive understanding, without which they cannot apply verification techniques in their daily works.

Hence the functional equivalence on continuations are too difficult as a criterion of program verification to be used by practitioners. As Mills' group aimed to establish a practical software development method, they had to stay within the direct semantics, which cannot handle the **goto** statement.

Hoare Logic, however, is based on the notion of predicate (on states) which is first order, just like that of a state transformer. If we adopt Hoare Logic as the base of the correctness proof, we can verify with-**goto** programs at a reasonable technical cost.

If we could use fully automated program verifiers in daily work, the discussion in the present work has no significance, but, in the current state of the art of program verification, we almost always must find loop invariants by ourselves. Moreover, simple and easily understandable loop invariants are undoubtedly very useful for making the maintenance of programs easier.

On program verification, there are several industrially accepted approaches, which are successfully applied to improve the reliability of commercial programs. Among others, Design by Contract by Meyer [42] is based on a limited (runtime-checkable) class of Hoare Logic. Mills and his colleague established Cleanroom Software Engineering whose verification is informal (but possibly rigorous) and it is widely known that its application to industrial software can decrease the number of bugs in programs by an order of magnitude [22, Section 1.5].

Though there has not yet been established any engineering-level software development method whose program verification process adopts full-scale Hoare Logic (or more powerful programming logic), it is possible to replace the base of the verification process of Cleanroom Software Engineering by Hoare Logic. Then, in such a development process, the simplicity and the intuitive clarity of loop invariants (and other assertions) are imperative and programming styles leading to such better loop invariants are undoubtedly important in engineering disciplines.

When we wrote, in Section 1, that Dijkstra's intention has hardly been studied with scientific bases, we did not forget that there are many works on various versions

of refinement calculi [3,49,54], program calculations [4,5,26,34] and other systematic programming methods [27,28,53]. All of them are direct successors of Dijkstra's own approach [18–21] and/or are formalized variants of Wirth's stepwise refinement [62,63] which itself has its origin in Dijkstra's structured programming.

It is clear that those formalized approaches to programming are very firmly based on programming logics. A problem of those formalized approaches and the informal stepwise refinement is that they force us to design programs in a *top-down* manner.

As Plauger points out in his essay, the program design method from finite state machine modeling is essentially *bottom-up*. Needless to say, this design method is widely accepted in industrial practice, and the light of programming logics should illuminate such a widely accepted engineering method. Our present work is a trial of lighting that practical design method with the most popular and classical programming logic, Hoare Logic.

Even though how designing and programming methods are important, they still are not purposes but only means. The ultimate goal of software production is to obtain easily quality-assurable (by correctness proofs) and easily understandable (so, easily maintainable) programs. Hence we must understand what program structures satisfy such nice properties with the light of programming logics. Gries [24] pointed out that the notion of structured programming should be considered from the viewpoint of program correctness. The present work is a trial of understanding structured programming along this direction, though it does not provide any explicit programming method, which Dijkstra stressed very much in his notes [17].

In the object-oriented paradigm, the idea of design pattern [23] has been highly praised, but actually known design patterns have only poor logical properties as we can see in the book [33] by Jézéquel and his colleagues. They tried to give contracts (in the sense of Design by Contract) to design patterns but, unfortunately, their contracts have rather poor logical contents.

As we discussed in [38], in order to establish science-based (truly mathematical) software engineering, it is essential to understand the logical properties of macro structures (design patterns, architectural styles, etc.) empirically found in real-world software.

Programming styles discussed in the present work are a class of such macro structures in software. They are still at a rather small scale but, at the same time, exceed the scale of programming language constructs.

Returning to the concrete cases discussed in our work, in the first case on escaping from nested loops, most programmers wisely use the with-**goto** style. On the other hand, in the second case on state transitions, many programmers may use the without-**goto** style, so Plauger had to write his essay to warn against such a common tendency. Our results give an affirmative account for the widely accepted programming style for the first case and a negative account (or a science-based warning) for the one in the second case. Therefore it is practically important and non-trivial to know what programming style is suitable for proving correctness.

In object-oriented programming, the flexibility (against future modifications) of programs seems to be regarded as the highest (or very high, at least) priority in programming, while, unfortunately, the correctness proof viewpoint seems to be hardly respected. We should return to the spirit of Dijkstra's structured programming and should

re-evaluate programming styles (and other macro-structures) with the measure of easiness of the correctness proof.

This work is only an initial and exceedingly small step in this direction. It is clear that many more studies are needed in analyzing programming styles (and other macro structures found in software) from the viewpoint of programming logics to make software engineering truly mathematical.

Acknowledgements

First of all, we are much obliged to the anonymous referee, who gave us significant and constructive critiques which makes our discussion in Section 5 more meaningful. We also wish to express our deepest thanks to the following people (in alphabetical order): Kenroku Nogi gave us constructive criticisms for an earlier version of this work and those criticisms were indispensable in completing the present work; Shigeru Otsuki gave us heartfelt encouragements without which we could not finish this work; Hirokazu Tanabe kindly told us that Plauger had already discussed the **goto** issue (though not logically but intuitively) on programming from finite state machine modeling in his essay; Hirokazu Yatsu gave many comments which were quite helpful in improving the present paper.

References

[1] K.R. Apt, E.-R. Olderog, Verification of Sequential and Concurrent Programs, 2nd ed., Springer-Verlag, 1997.
[2] E. Ashcroft, Z. Manna, The translation of 'go to' programs to 'while' programs, in: Proceedings of 1971 IFIP Congress, vol. 1, North-Holland, 1972, pp. 250–255. Reprinted in: [59], pp. 51–61.
[3] R.-J. Back, J. von Wright, Refinement Calculus, Springer-Verlag, 1998.
[4] R.C. Backhouse, Program Construction and Verification, Prentice-Hall, 1986.
[5] R.C. Backhouse, Program Construction: Calculating Implementations from Specifications, John-Wiley, 2003.
[6] J.W. de Bakker, Mathematical Theory of Program Correctness, Prentice-Hall, 1980.
[7] V.R. Basili, F.T. Baker, Tutorial on Structured Programming: Integrated Practices, IEEE Computer Society Press, 1981.
[8] S.A. Becker, J.A. Whittaker, Cleanroom Software Engineering Practices, Idea Group Publishing, 1997.
[9] C. Böhm, G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, Commun. ACM 9 (5) (1966) 366–371. Reprinted in: [59], pp. 13–25.
[10] A. de Bruin, **Goto** statements: Semantics and deduction systems, Acta Inform. 15 (1981) 385–424.
[11] M. Clint, C.A.R. Hoare, Program proving: Jumps and functions, Acta Inform. 1 (1972) 214–224.
[12] D.C. Cooper, Böhm and Jacopini's reduction of flow charts, Commun. ACM 10 (1967) 463, 467. Reprinted in: [60], pp. 205–206.
[13] D.C. Cooper, Some transformations and standard forms of graphs, with applications to computer programs, in: E. Dale, D. Michie (Eds.), Machine Intelligence, vol. 2, Edinburgh at the University Press, 1967, pp. 21–32.
[14] O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming, Academic Press, 1972.
[15] E.W. Dijkstra, Goto statement considered harmful, Commun. ACM 11 (1968) 147–148. Reprinted in: [59], pp. 29–33; and [61], pp. 102–103.
[16] E.W. Dijkstra, Structured programming, in: J.N. Buxton, B. Randell (Eds.), Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, 27–31 October 1969, NATO Science Committee, Rome, Italy, 1970, pp. 84–88. Reprinted in: J.N. Buxton, P. Naur, B. Randell

(Eds.), Software Engineering: Concepts and Techniques, Petrocelli/Charter, 1976, pp. 222–226; [59], pp. 43–48; and [7], pp. 38–41.
[17] E.W. Dijkstra, Notes on Structured Programming, in: [14], 1972, pp. 1–82 (Chapter 1).
[18] E.W. Dijkstra, Guarded commands, nondeterminacy, and formal derivation of programs, Commun. ACM 18 (1975) 453–457. Reprinted in: [25], pp. 166–175; [58], pp. 233–242; and [61], pp. 165–169.
[19] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.
[20] E.W. Dijkstra, W.H.J. Feijen, A Method of Programming, Prentice-Hall, 1988.
[21] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Springer-Verlag, 1990.
[22] M. Dyer, The Cleanroom Approach to Quality Software Development, John Wiley & Sons, 1992.
[23] E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[24] D. Gries, On structured programming, Commun. ACM 17 (1974) 655–657. Reprinted in: [25], pp. 70–74.
[25] D. Gries (Ed.), Programming Methodology: A Collection of Articles by Members of IFIP WG2.3, Springer-Verlag, 1978.
[26] D. Gries, The Science of Programming, Springer-Verlag, 1981.
[27] E.C.R. Hehner, The Logic of Programming, Prentice-Hall, 1984.
[28] E.C.R. Hehner, A Practical Theory of Programming, Springer-Verlag, 1993.
[29] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580, 583. Reprinted in: [30], pp. 45–58; [25], pp. 89–100; and [61], pp. 500–505.
[30] C.A.R. Hoare, in: C. Jones (Ed.), Essays in Computing Science, Prentice-Hall, 1989.
[31] C.A.R. Hoare, N. Wirth, An axiomatic definition of programming language PASCAL, Acta Inform. 2 (1973) 335–355. Reprinted in: [30], pp. 153–169; and [61], pp. 506–526.
[32] M.E. Hopkins, A case for the goto, in: Proceedings of the 25th National ACM Conf., 1972, pp. 787–790. Reprinted in: [59], pp. 101–109.
[33] J.-M. Jézéquel, M. Train, C. Mingins, Design Patterns and Contracts, Addison-Wesley, 2000.
[34] A. Kaldewaij, Programming: The Derivation of Algorithms, Prentice-Hall, 1990.
[35] A.J. Kfoury, R.N. Moll, M.A. Arbib, A Programming Approach to Computability, Springer-Verlag, 1982.
[36] D.E. Knuth, Structured programming with go to statements, Comput. Surv. 6 (1974) 260–301. Reprinted in: [58], pp. 140–194; [59], pp. 259–321 and [61], pp. 104–144.
[37] D.E. Knuth, R.W. Floyd, Notes on avoiding 'Go to' statements, Inform. Process. Lett. 1 (1971) 23–31. Reprinted in: [60], pp. 153–162.
[38] H. Kondoh, What is *'Mathematicalness'* in software engineering? in: T. Maibaum (Ed.), Fundamental Approaches to Software Engineering, in: Lect. Not. Comput. Sci., vol. 1783, Springer-Verlag, 2000, pp. 163–177.
[39] B. Leavenworth (Ed.), Control Structures in Programming Languages—The GO TO Controversy—Debate, ACM SIGPLAN Notices 7 (11) (1972), 53–91.
[40] R.C. Linger, H.D. Mills, On the Development of Large Reliable Programs, in: [58], 1977, pp. 120–139.
[41] R.C. Linger, H.D. Mills, B.I. Witt, Structured Programming: Theory and Practice, Addison-Wesley, 1979.
[42] B. Meyer, Object-Oriented Software Construction, 2nd ed., Addison-Wesley, 1997.
[43] H.D. Mills, Top down programming in large systems, in: R. Rustin (Ed.), Debugging Techniques in Large Systems, Prentice Hall, 1971, pp. 41–55. Reprinted in: [46], pp. 91–101.
[44] H.D. Mills, Mathematical foundations of structured programming, IBM Report, FSC 72-6012, 1972, pp. 18–83. Reprinted in: [7], pp. 42–107; [46], pp. 115–179; and [60], pp. 220–262.
[45] H.D. Mills, The new math of computer programming, Commun. ACM 18 (1975) 43–48. Reprinted in: [46], pp. 215–230.
[46] H.D. Mills, Software Productivity, Dorset House, 1988.
[47] H.D. Mills, V.R. Basili, J.D. Gannon, R.G. Hamlet, Principles of Computer Programming: A Mathematical Approach, Wm. C. Brown Publishers, 1986.
[48] H.D. Mills, R.C. Linger, A.R. Hevner, Principles of Information Systems Analysis and Design, Academic Press, 1986.
[49] C. Morgan, Programming from Specification, 2nd ed., Prentice Hall, 1994.
[50] J.H. Poore, C.J. Trammell (Eds.), Cleanroom Software Engineering: A Reader, NCC Blackwell, 1996.
[51] S.J. Prowell, C.J. Trammell, R.C. Linger, J.H. Poore, Cleanroom Software Engineering: Technology and Process, Addison-Wesley, 1999.
[52] P.J. Plauger, Programming on Purpose: Essays on Software Design, Prentice-Hall, 1993.

[53] J.C. Reynolds, The Craft of Programming, Prentice-Hall, 1981.
[54] W.-P. de Roever, K. Engelhardt, Data Refinement: Model-Oriented Proof Methods and their Comparison, Cambridge University Press, 1998.
[55] A.M. Stavely, Toward Zero-Defect Programming, Addison-Wesley, 1999.
[56] C. Strachey, C.P. Wadsworth, Continuations: A mathematical semantics for handling full jumps, Technical Monograph PRG-11, Oxford University Computing Laboratory, 1974.
[57] R.D. Tennent, Principles of Programming Languages, Prentice-Hall, 1982.
[58] R.T. Yeh (Ed.), Current Trends in Programming Methodology, Software Specification and Design, vol. I, Prentice-Hall, 1977.
[59] E. Yourdon (Ed.), Classics in Software Engineering, Yourdon Press, 1979.
[60] E. Yourdon (Ed.), Writings of the Revolution: Selected Readings on Software Engineering, Yourdon Press, 1982.
[61] A.I. Wasserman (Ed.), Tutorial: Programming Language Design, IEEE Computer Society Press, 1980.
[62] N. Wirth, Program development by stepwise refinement, Commun. ACM 14 (1971) 221–227. Reprinted in: [25], pp. 321–335; also [60], and pp. 99–111.
[63] N. Wirth, Systematic Programming — An Introduction, Prentice-Hall, 1973.
[64] W.A. Wulf, A case against the goto, in: Proceedings of the 25th National ACM Conf., 1972, pp. 791–797. Reprinted in: [59], pp. 85–98.

To `Goto` Where No Statement Has Gone Before

Mike Barnett and K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA, {mbarnett,leino}@microsoft.com

Abstract. This paper presents a method for deriving an expression from the low-level code compiled from an expression in a high-level language. The input is the low-level code represented as blocks of code connected by `goto` statements, *i.e.*, a control flow graph (CFG). The derived expression is in a form that can be used as input to an automatic theorem prover. The method is useful for program verification systems that take as input both programs and specifications after they have been compiled from a high-level language. This is the case for systems that encode specifications in an existing programming language and do not have a special compiler. The method always produces an expression, unlike the heuristics for decompilation which may fail. It is efficient: the resulting expression is linear in the size of the CFG by maintaining all sharing of subgraphs.

0 Introduction

A program verifier checks that a given program satisfies its specifications. Some programming languages such as Eiffel [16], Java with JML [12], or Spec# [3] provide the programmer a nice syntax for writing the specifications in the source text. This has many advantages, *e.g.*, that programmers are immediately aware of the relationship between their code and its specification. However, in a multi-language platform like .NET, one would like to have one program verifier that works for any language, regardless of what special syntax, if any, each language may provide. In this paper, we consider one issue that arises in such a multi-language setting.

Code Contracts for .NET [1] is a library-based framework for writing specifications in .NET code. Programmers use the methods from the contract library to write specifications within their program (written in any .NET language, like C#, Visual Basic, or F#) as stylized method calls at the beginning of a method's body. For example, Figure 0 shows a method with a postcondition, expressed as a call to `Contract.Ensures`. The regular .NET compiler for the source program is invoked to produce bytecode. Code Contracts then has several tools which operate on the resulting bytecode, for example the runtime checker rewrites the bytecode to move the evaluation of postconditions to all of the method body's exit points.

We are connecting an existing program verifier to the Code Contracts framework by translating the compiled bytecode into an intermediate verification language, Boogie 2 [0,15, 13], and then generating verification conditions for a theorem prover (we primarily use the SMT solver Z3 [7]). Source-program uses of Code Contracts show up in the bytecode as calls to the contract methods, preceded by a snippet of code that evaluates the arguments. For the example in Figure 0, the bytecode computes the postcondition and then passes that boolean value as the argument to `Contract.Ensures`.

Therefore, *expressions* in the source language become *code*. In general, the code is a linearized form of a DAG, with a high degree of sharing.

The problem is that the verification conditions needed by the theorem prover must be first-order formulas. While there are various contexts in which this can be avoided, the body of a quantifier *must* be a genuine expression, not code.

We propose to convert the code representing a boolean expression back into a genuine expression in two steps. First, our program verifier identifies the code snippets in the bytecode and converts them into *code expressions* of the form

$$\{\{ \quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ e \quad \}\}$$

where S denotes some code in the intermediate verification language, e denotes the value returned by the code expression, and b denotes a list of local variables that may be used in S and e. Defining code expressions in the intermediate verification language has the advantage that we can make use of facilities in the intermediate verification language that expect expressions, like pre- and postconditions and bodies of logical quantifiers.

Second, we define the meaning of a code expression in terms of a first-order formula. We show how to construct this formula from the code expression. The resulting formula is "efficient": it maintains the sharing in the DAG, and is thus linear in the size of the control-flow graph of the code expression.

In this paper, we also give some healthiness conditions for what it means to interpret code as a genuine expression.

1 The Starting Point

An example program in the C# programming language using Code Contracts is shown in Figure 0. The example shows a simple method that has a postcondition (encoded

```
using System.Diagnostics.Contracts;
public class C {

  public int[] M(int[] A, int k) {
    Contract.Ensures(
      Contract.Result<int[]>().Length == A.Length &&
      Contract.ForAll(
        0, Contract.Result<int[]>().Length,
        i => A[i] == 0 || Contract.Result<int[]>()[i] == k/A[i]
      )
    );
    ...
  }
}
```

Fig. 0. A portion of a C# program using Code Contracts

using the method `Contract.Ensures`). It states that the return value (encoded with `Contract.Result`) has the same length as the parameter `A` and that each element is the division of `k` by the corresponding element of `A`, except in the case that the element is zero [0]. In order to state that, it uses a quantifier: the method `Contract.Forall` is given three arguments, an inclusive lower bound, an exclusive upper bound, and an *anonymous delegate*. The latter is the .NET form for a *lambda expression*, *i.e.*, a functional value. The type of `Forall` restricts the function to take a single argument of type **`int`** and return a boolean[1]. In the example, the function's parameter is named `i`. In traditional notation, the function would be written as $(\lambda\ i : int\ .\ A[i] \neq 0 \Rightarrow result[i] = k/A[i])$. Anonymous delegates are lexically scoped and "capture" references, such as to the method's parameter `A`.

The source-language compiler (in this case the C# compiler) is used to compile the program to MSIL. Since we do not have control over the C# compiler, the specifications are compiled into MSIL just as the "real" program is. In particular, short-circuit boolean expressions are compiled into *code expressions*. These are a linearized DAG of basic blocks with assignment statements and goto statements where the value of the boolean expression is left on the stack. For the current example, Figure 1 shows the MSIL that the anonymous delegate in Figure 0 compiles into. A more readable form written in C# is:

```
bool Anonymous(int i) {
  bool b;
  if (A[i] == 0) goto L_0024;
  if (result[i] == k/A[i]) goto L_0024;
  b := false;
  goto L_0028;
  L_0024: b := true;
  L_0028: return b;
}
```

2 The Midpoint: Boogie

An intermediate verification language serves a purpose analogous to that of an intermediate representation in a compiler: it separates the concerns of defining source-language semantics from the concerns of generating formulas for a theorem prover. Many program verifiers are built around an architecture that uses an intermediate verification language (*e.g.*, [0,10, 5]).

[0] The method `Contract.Result` is generic and must be instantiated since its type cannot be inferred from its arguments because it is a nullary method (hence the open-close parentheses). Type instantiation is indicated by referring to the return type of the method, **`int`**`[]`, within angled brackets. This shows why it is so nice to have a language provide surface syntax for specifications!

[1] There is another version of `Forall` that allows a more general predicate.

```
.method public hidebysig instance bool <M>b__0(int32 i) cil managed
{
  .maxstack 4
  .locals init (
    [0] bool CS$1$0000)
  L_0000: ldarg.0
  L_0001: ldfld int32[] C/<>c__DisplayClass1::A
  L_0006: ldarg.1
  L_0007: ldelem.i4
  L_0008: brfalse.s L_0024
  L_000a: call !!0 [Microsoft.Contracts]System.Diagnostics.Contracts.Contract::Result<int32[]>()
  L_000f: ldarg.1
  L_0010: ldelem.i4
  L_0011: ldarg.0
  L_0012: ldfld int32 C/<>c__DisplayClass1::k
  L_0017: ldarg.0
  L_0018: ldfld int32[] C/<>c__DisplayClass1::A
  L_001d: ldarg.1
  L_001e: ldelem.i4
  L_001f: div
  L_0020: ceq
  L_0022: br.s L_0025
  L_0024: ldc.i4.1
  L_0025: stloc.0
  L_0026: br.s L_0028
  L_0028: ldloc.0
  L_0029: ret
}
```

Fig. 1. The bytecode compiled from the body of the anonymous delegate in Figure 0. The labels on each line are the byte offsets of the instructions. The code from offset 0x0 to 0x7 represents the left disjunct $A[i] == 0$. The right disjunct, `Contract.Result<int[]>()[i] == k/A[i]`, is computed in the code from offset 0x0a to 0x20. "arg 0" refers to **`this`**, the implicit receiver and "arg 1" refers to the parameter `i`. There is an implicit receiver because the captured variables in an anonymous delegate become fields on a compiler-generated class in order to retain the necessary state in between invocations. In this case, there are fields for `A` and `k`.

2.0 Previously...

We reiterate the language from [2], which forms the core of the Boogie intermediate verification language:

$$
\begin{array}{lll}
\mathit{Program} & ::= & \mathit{Block}^{+} \\
\mathit{Block} & ::= & \mathit{BlockId}\ :\ \mathit{Stmt}\ ;\ \mathit{Goto} \\
\mathit{Stmt} & ::= & \mathit{VarId} := \mathit{Expr}\ \mid\ \mathbf{havoc}\ \mathit{VarId} \\
 & & \mid\ \mathit{Stmt}\ ;\ \mathit{Stmt}\ \mid\ \mathbf{skip} \\
 & & \mid\ \mathbf{assert}\ \mathit{Expr}\ \mid\ \mathbf{assume}\ \mathit{Expr} \\
\mathit{Goto} & ::= & \mathbf{goto}\ \mathit{BlockId}^{*}
\end{array}
$$

In our core language, a program consists of a set of basic blocks, where the unstructured control flow between blocks is given by goto statements. A goto with multiple target

labels gives rise to a non-deterministic choice; a goto with no target labels gives rise to normal termination. The $BlockId$'s listed in a goto statement are the *successors* of the block.

The semantics of the core language is defined over *traces*, *i.e.*, sequences of program states. Each finite trace either terminates normally or ends in an error. There are two assignment statements: $x := e$ sets variable x to the value of expression e, and $\mathbf{havoc}\ x$ sets x to an arbitrary value. Semi-colon is the usual sequential composition of statements, and $\mathbf{skip}$, which is the unit element of semi-colon, terminates normally without changing the state. The assert statement $\mathbf{assert}\ e$ behaves as $\mathbf{skip}$ if e evaluates to $true$; otherwise, it causes the trace to end in an error (we say the trace *goes wrong*). The assume statement $\mathbf{assume}\ e$ is a *partial command* [18]: it behaves as $\mathbf{skip}$ if e evaluates to $true$; otherwise, it leads to no traces at all. The assume statement is thus used to describe which traces are feasible.

The normally terminating traces of a block are extended with the traces of the block's successors.

Note that the core language does not have a method call as a primitive statement; a method call is encoded as a sequence of statements that assert the method's precondition, use havoc statements to set the locations that the method may modify to an arbitrary value, and then assume the method's postcondition.

Verification condition generation proceeds by first converting the program into *passive form*, where loops are cut (see [2]) and where all assignment statements are replaced by assumptions expressed over a single-assignment form of the program variables [11]. For example, a statement

$$x := y\ ;\ x := x + y\ ;\ \mathbf{assert}\ y < x$$

is converted into a passive form like

$$\mathbf{assume}\ x_1 = y_0\ ;\ \mathbf{assume}\ x_2 = x_1 + y_0\ ;\ \mathbf{assert}\ y_0 < x_2$$

where y_0, x_1, and x_2 are fresh variables.

The passive program is turned into a formula via *weakest preconditions* [9]. For any passive statement S and any predicate Q characterizing a set of post-states of S, $wp[\![S, Q]\!]$ is a predicate that characterizes those pre-states from which execution of S will not go wrong and will end in a state described by Q. The weakest-precondition equations for passive statements are as follows:

$$\begin{array}{lcl} wp[\![\mathbf{skip}, Q]\!] & = & Q \\ wp[\![S\ ;\ T,\ Q]\!] & = & wp[\![S, wp[\![T, Q]\!]]\!] \\ wp[\![\mathbf{assert}\ e, Q]\!] & = & e \wedge Q \\ wp[\![\mathbf{assume}\ e, Q]\!] & = & e \Rightarrow Q \end{array}$$

In each of the last two equations, the occurrence of e on the left-hand side is an expression in Boogie, whereas its occurrence on the right-hand side must be an expression in the input language of the theorem prover. These expressions are usually so similar that we do not mind glossing over this difference; however, for code expressions this makes an important difference.

To deal with (unstructured) control flow, we introduce a variable A_{ok} for every block labeled A, and we define A_{ok} to be $true$ iff no execution from A goes wrong [2]. In particular, for any block A with body S and successors $Succ(A)$, we define

$$A_{ok} \quad = \quad wp[\![S, \bigwedge_{B \in Succ(A)} B_{ok}]\!]$$

2.1 Adding Code Expressions to Boogie

We extend the core language to include code expressions. Previously [2], we left implicit the definition of *Expr* (and its implementation did not allow code expressions). Now, we explicitly extend the definition of *Expr* to include them:

$$\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Expr}\ \mathit{op}\ \mathit{Expr} \mid \mathit{MethodCall} \mid \mathit{CodeExpr} \\
\mathit{CodeExpr} & ::= & \{\!\{\ \mathit{LocalDecl}^*\ \mathit{CodeBlock}^+\ \}\!\} \\
\mathit{LocalDecl} & ::= & \mathit{VarId}\ :\ \mathit{Type} \\
\mathit{CodeBlock} & ::= & \mathit{BlockId}\ :\ \mathit{Stmt}\ ;\ \mathit{Transfer} \\
\mathit{Transfer} & ::= & \mathit{Goto} \mid \mathit{Return} \\
\mathit{Return} & ::= & \mathbf{return}\ \mathit{Expr}
\end{array}$$

We need each code expression to be a self-contained unit. In order to achieve that, we assume that each code expression is *well-formed* by meeting the following conditions:

- A *transfer command* comprising a goto statement has at least one successor.
- All successors are other blocks within the code expression.
- No block in a code expression is a successor of any block not in the code expression.
- The graph induced on the blocks by the successor relation is acyclic.
- All paths within the code expression end with a block whose transfer command comprises a return statement.

We also assume each code expression has a first block labeled "*Start*", which is the entry point to the code expression.

2.2 When Is Code an Expression?

It is one thing to syntactically allow code expressions in Boogie, but we still must consider when a code expression really does represent a genuine expression, *i.e.*, when we are justified in using the same semantics for them as for genuine expressions. Thus the question of this section: when can we look at a chunk of code and consider it a genuine expression?

It must meet four requirements:

0. It must be deterministic. (All branches are mutually exclusive.)
1. It must be total in terms of not being a partial command.
2. It must be total, in the "expression sense". That is, its execution does not go wrong (*i.e.*, failing an assertion, like dereferencing *null* or dividing by zero).

3. It must not have any side effects (on variables other than the local variables it introduces).

In our setting, the first two requirements are satisfied since our code expressions are the output of a .NET compiler. That is, .NET bytecode (IL) obeys Dijkstra's "Law of the Excluded Miracle" [9] and does not contain any non-deterministic features. (If we were to allow code expressions to contain partial commands, then our scheme derives the value $true$ in states where the partiality comes into play.)

We enforce the third requirement by omitting all assertions within a code expression. Such *definedness checks*, *e.g.*, that a divisor is non-zero, are enforced by many verifiers [14] for expressions separately from the expressions themselves by inserting extra checks which guarantee that the expression is total.

The fourth requirement is enforced by making sure that all assignment statements within a code expression are to its local variables and that all method calls are to *pure methods*, *i.e.*, methods whose Boogie encoding do not have any modifies clauses.

3 The Endpoint: Deriving an Expression from Code

But now we have a mismatch: we have code expressions in places where they need to be translated to expressions in the prover's language. We either need a new definition for the weakest precondition when an expression is a code expression or we need a translation scheme that produces a genuine expression from a code expression.

We take the latter approach and, for now, restrict ourselves to *boolean* code expressions, *i.e.*, the value they return is a boolean. For boolean code expressions that meet the requirements in Section 2.2, we compute an equivalent boolean expression (that does not contain any code expressions). For the code expression

$$\{\{ \quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ e \quad \}\}$$

the equivalent boolean expression is

$$(\ \forall\ b\ \bullet\ \ wp[\![S, e]\!]\ ) \tag{0}$$

This presumes that S is a structured command, *i.e.*, control flows from S to the return statement. When the code expression is unstructured, then we form the block equations as in Section 2.0. The only difference is that for any block A whose transfer statement comprises a return statement $\mathbf{return}\ e$, we define the block equation as:

$$A_{ok} \quad = \quad wp[\![S, e]\!]$$

Because code expressions are acyclic, we can avoid having to quantify over the block variables by defining them via let-expressions. (Z3 supports the SMT-LIB format [19], which allows let-expressions in the verification condition.)

So the body of the anonymous delegate can be represented in Boogie as:

$\{\{\ \ b\colon \mathbf{bool};$
$\mathit{Start} : \mathbf{skip}\ ;\ \mathbf{goto}\ L0, L1;$
$L0 : \mathbf{assume}\ A[i] = 0\ ;\ \mathbf{goto}\ L2;$
$L1 : \mathbf{assume}\ A[i] \neq 0\ ;\ \mathbf{goto}\ L3, L4;$
$L2 : b := \mathit{true}\ ;\ \mathbf{goto}\ L5;$
$L3 : \mathbf{assume}\ \mathit{result}[i] = k/A[i]\ ;\ \mathbf{goto}\ L2;$
$L4 : \mathbf{assume}\ \mathit{result}[i] \neq k/A[i]\ ;\ b := \mathit{false}\ ;\ \mathbf{goto}\ L5;$
$L5 : \mathbf{skip}\ ;\ \mathbf{return}\ b;\ \ \}\}$

We first convert it into passive form by introducing a new *incarnation* of a variable each time it is assigned. Join points (*e.g.*, $L5$) also produce a new incarnation with equations pushed into each predecessor relating the value of the variable in that branch with that of the join point's incarnation.

$\{\{\ \ b\colon \mathbf{bool};$
$\mathit{Start} : \mathbf{skip}\ ;\ \mathbf{goto}\ L0, L1;$
$L0 : \mathbf{assume}\ A[i] = 0\ ;\ \mathbf{goto}\ L2;$
$L1 : \mathbf{assume}\ A[i] \neq 0\ ;\ \mathbf{goto}\ L3, L4;$
$L2 : \mathbf{assume}\ b_0 = \mathit{true}\ ;\ \mathbf{assume}\ b_2 = b_0\ ;\ \mathbf{goto}\ L5;$
$L3 : \mathbf{assume}\ \mathit{result}[i] = k/A[i]\ ;\ \mathbf{goto}\ L2;$
$L4 : \mathbf{assume}\ \mathit{result}[i] \neq k/A[i]\ ;\ \mathbf{assume}\ b_1 = \mathit{false}\ ;\ \mathbf{assume}\ b_2 = b_1\ ;\ \mathbf{goto}\ L5;$
$L5 : \mathbf{skip}\ ;\ \mathbf{return}\ b_2;\ \ \}\}$

Then the block equations, written as let-expressions, are:

$$
\begin{array}{llll}
\mathbf{let} & L5_{ok} & = & wp[\![\mathbf{skip}, b_2]\!] \quad \mathbf{in} \\
\mathbf{let} & L2_{ok} & = & wp[\![\mathbf{assume}\ b_0 = \mathit{true}\ ;\ \mathbf{assume}\ b_2 = b_0, L5_{ok}]\!] \quad \mathbf{in} \\
\mathbf{let} & L3_{ok} & = & wp[\![\mathbf{assume}\ \mathit{result}[i] = k/A[i], L2_{ok}]\!] \quad \mathbf{in} \\
\mathbf{let} & L4_{ok} & = & wp[\![\mathbf{assume}\ \mathit{result}[i] \neq k/A[i]\ ;\ \mathbf{assume}\ b_1 = \mathit{false}\ ; \\
 & & & \mathbf{assume}\ b_2 = b_1, L5_{ok}]\!] \quad \mathbf{in} \\
\mathbf{let} & L1_{ok} & = & wp[\![\mathbf{assume}\ A[i] \neq 0, L3_{ok} \wedge L4_{ok}]\!] \quad \mathbf{in} \\
\mathbf{let} & L0_{ok} & = & wp[\![\mathbf{assume}\ A[i] = 0, L2_{ok}]\!] \quad \mathbf{in} \\
\mathbf{let} & \mathit{Start}_{ok} & = & wp[\![\mathbf{skip}, L0_{ok} \wedge L1_{ok}]\!] \quad \mathbf{in} \\
 & \mathit{Start}_{ok} & &
\end{array}
$$

After simplifying[2] the expression is equivalent to:

$$
\begin{array}{llll}
\mathbf{let} & L5_{ok} & = & b_2 \quad \mathbf{in} \\
\mathbf{let} & L2_{ok} & = & b_0 = \mathit{true} \Rightarrow b_2 = b_0 \Rightarrow L5_{ok} \quad \mathbf{in} \\
\mathbf{let} & L3_{ok} & = & \mathbf{result}[i] = k/A[i] \Rightarrow L2_{ok} \quad \mathbf{in} \\
\mathbf{let} & L4_{ok} & = & \mathbf{result}[i] \neq k/A[i] \Rightarrow b_1 = \mathit{false} \Rightarrow b_2 = b_1 \Rightarrow L5_{ok} \quad \mathbf{in} \\
\mathbf{let} & L1_{ok} & = & A[i] \neq 0 \Rightarrow L3_{ok} \wedge L4_{ok} \quad \mathbf{in} \\
\mathbf{let} & L0_{ok} & = & A[i] = 0 \Rightarrow L2_{ok} \quad \mathbf{in} \\
\mathbf{let} & \mathit{Start}_{ok} & = & L0_{ok} \wedge L1_{ok} \quad \mathbf{in} \\
 & \mathit{Start}_{ok} & &
\end{array}
$$

[2] Yes, we realize it doesn't look particularly simple. We mean that we have applied the definition of the weakest-precondition.

If we denote that entire expression by R, then the genuine expression which is equivalent to the body of the anonymous delegate is:

$$(\forall\, b_0, b_1, b_2 \bullet\ R\ )$$

and the entire postcondition of the method in Figure 0 is:

$$\begin{array}{l} \mathbf{result}.Length = A.Length\ \ \wedge \\ (\forall\, i \bullet\ 0 \leq i < \mathbf{result}.Length \Rightarrow (\forall\, b_0, b_1, b_2 \bullet\ R\ )) \end{array}$$

Looking closely[3], one can see that the truth value of this expression is equivalent to the original postcondition.

We perform this translation in a depth-first traversal of the program, replacing each code expression from innermost to outermost.

4 Non-Boolean Code Expressions

In this section, we extend our translation of boolean code expressions to code expressions of any type. The basic idea is to distribute the non-boolean code expression to a context where its value can be stated as a boolean antecedent.

Let $G[\cdot]$ denote an expression context with a "hole". That is, if we place an expression e in the hole, written $G[e]$, we get an expression with an occurrence of e as a subexpression. We assume bound variables in G are suitably renamed so as to always avoid name capture of the free variables of e.

Now, let e be a code expression of an arbitrary type (that is, not necessarily boolean), and let $VC[e]$ be the verification condition (in other words, the verification condition contains an occurrence of e). We now show how to transform expression $VC[e]$ to an equivalent expression that does not contain this occurrence of e but instead contains a boolean code expression. First, for any variable x occurring free in e and introduced in the verification condition by a let binding $\mathbf{let}\ x = t\ \mathbf{in}\ u$, replace x by t in e. Then, consider any context G such that $G[e]$ is a boolean subexpression of $VC[e]$; that is, $G[e]$ is some subexpression of $VC[e]$ such that the free variables of e are also free variables of $G[e]$. Specifically, if e is contained in a quantifier, then $G[e]$ can be the body of the innermost such quantifier; if e is not contained in any quantifier, then $G[e]$ can simply be $VC[e]$.

Since $G[e]$ is boolean, it is equivalent to the expression

$$(\forall\, k \bullet\ k = e \Rightarrow G[k]\ )$$

where k is a fresh variable. Considering that e is a code expression, we have:

$$\begin{array}{ll} & (\forall\, k \bullet\ k = \{\!\{\quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ d\quad \}\!\} \Rightarrow G[k]\ ) \\ = & \quad \{\ \text{distribute "}k =\text{" over the code expression}\ \} \\ & (\forall\, k \bullet\ \{\!\{\quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ k = d\quad \}\!\} \Rightarrow G[k]\ ) \end{array}$$

The transformation we have just showed can thus be used to replace non-boolean code expressions with boolean ones, after which the semantics that we have defined for boolean code expressions earlier in the paper can be used.

[3] Squinting helps too.

5 Related Work

An alternative means for recovering boolean expressions would be to *decompile* the MSIL back into a high-level expressions [6]. For the trivial example with which we have demonstrated our scheme, this clearly would be quite easy.

However, we believe all decompilers are heuristic and so may not always be able to successfully decompile an expression, certainly not without perhaps introducing the same redundancy as a tree-encoding of the DAG, compared to the linear size of our derived expression. Also, a decompiler's goal is to produce an expression which is "close" to a boolean expression that a programmer would write. We are not concerned with making the expression "readable", but instead just need to be able to communicate it to a theorem prover.

There are other approaches that derive a functional form, *i.e.*, an expression, from imperative code [17, 4], but it isn't clear whether their results are more usable by an SMT solver than ours. It also isn't clear whether the sharing represented in the compiled CFG is preserved.

6 Conclusions

In Section 0, we noted that there are contexts in which code expressions do not need to be converted back into a genuine expression. For instance, Boogie encodes preconditions (respectively, postconditions) as assume (respectively, assert) statements in the Boogie program itself. Instead of forming the verification condition $P \Rightarrow wp[\![S, Q]\!]$ for a program S, precondition P, and postcondition Q, it computes the weakest precondition with respect to $true$ of the program:

$$\begin{array}{l}\textbf{assume}\ \ P\ ;\\ S;\\ \textbf{assert}\ \ Q\end{array}$$

This means that if P or Q are code expressions, they can be *in-lined* and the assume (assert) "distributed" so that any return statement in the code expression, $\textbf{return}\ e$, becomes an assertion (assumption) on e. Then, the definitions of wp in Section 2 will produce a first-order formula that is accepted by theorem provers.

But this cannot be done for quantifiers: instead they must be translated into an equivalent quantifier in the input language of the theorem prover, which does not include code expressions. Therefore, we need to perform our technique only for code expressions occurring within a quantifier. As we progress with the implementation of this scheme in Boogie, we will need to see if the introduction of the quantifier in Equation 0 leads to problems with triggering. (A trigger is the pattern a Simplify-like SMT solver requires before it instantiates a quantifier [8].)

The general form for a quantifier is:

$$(\ Q\, x : T \mid\ x \in D \bullet\ P(x)\ )$$

It has four parts that are defined by the specification language: Q, T, D, and P. The kind of the quantifier is Q, which is usually either universal or existential quantification. T is the type of the bound variable x. Different specification languages also may

restrict the kind of domain, D, that a bound variable may be drawn from. For instance, Spec# restricts D to be a finite, computable set that has an interpretation at runtime. Our technique is concerned only with the fourth part: representing the body $P(x)$. The choices made for the other three are completely orthogonal.

In summary, in this paper, we have adapted our previous work on verification condition generation [2] to provide a scheme for turning code that represents an expression back into an expression in order for it to be easily translated into input for an automatic theorem prover. The scheme avoids decompilation and is efficient. We also outlined four healthiness conditions for ensuring that a code expression can be treated as a genuine expression.

Acknowledgements

We would like to thank Manuel Fähndrich, Francesco Logozzo, and Michał Moskal for valuable help and insight. Comments by the anonymous referees were also helpful.

References

0. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
1. Mike Barnett, Manuel Fähndrich, and Francesco Logozzo. Embedded contract languages. In *ACM SAC - OOPS*. ACM, March 2010.
2. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM Press.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
4. Arthur Charguéraud. Program verification through characteristic formulae. In *ACM SIGPLAN International Conference on Functional Programming*, 2010. To appear.
5. Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS 2007*, pages 19–33, 2007.
6. Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software — Practice and Experience*, 25(7):811–829, July 1995.
7. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.
8. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
9. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

10. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, July 2007.
11. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.
12. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
13. K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at http://research.microsoft.com/ leino/papers.html.
14. K. Rustan M. Leino. Specification and verification of object-oriented software. In Manfred Broy, Wassiou Sitou, and Tony Hoare, editors, *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 231–266. IOS Press, 2009. Summer School Marktoberdorf 2008 lecture notes.
15. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, March 2010.
16. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.
17. Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *FMCAD*, pages 1–8, 2008.
18. Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.
19. Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

A Formal Basis For Removing *Goto* Statements

SI PAN AND R. GEOFF DROMEY

Software Quality Institute, Griffith University, Brisbane, QLD. 4111, Australia
Email: G.Dromey@cit.gu.edu.au

***Goto* statements detract from the quality of imperative programs. They tend to make control-structures difficult to understand and, at the same time, introduce the risk of non-termination and other correctness problems. A new, formal, generally applicable procedure for removing all *goto* statements from program structures is presented. This method is based on formal semantics and congruent equivalence transformations. Not only does the method logically simplify program structures; it also detects a range of defects including a class of non-termination problems, unreachable code and redundancy problems. The method can also be used to eliminate recursion.**

Received March 29, 1994; revised March 20, 1996

1. INTRODUCTION

Imperative program components containing ***goto*** statements are usually regarded as very difficult structures to analyse, modify, restructure and prove correct [1]. It has long been recognized that the use of ***goto*** statements significantly detracts from the structural integrity, simplicity, reliability and the ultimate quality of programs.

Although the use of ***goto*** statements became unfashionable more than two decades ago, they are still found in some programs that must be re-engineered and maintained. An examination of the literature in this area over the past two decades reveals that a number of studies [2–11] have been done on methods for eliminating ***goto*** statements and recursion [12, 13] from programs. Assessing these transformations, methods and results we conclude that:

- those that involve transformations heavily rely on pattern matching and therefore tend to lack generality;
- they sometimes result in transformations that change properties of the original programs;
- they often introduce logical and textual inefficiencies.

There appears to be no powerful, widely applicable, formal means for removing ***goto*** statements from programs. What we will describe here is a new, formally based, systematic method for removing ***goto*** statements from sequences and loop structures. In our treatment we assume a ***goto*** statement always has associated with it a ***label*** which determines the next statement to be executed when the ***goto*** is reached.

The proposed method has many similarities to the way we use substitution to solve a set of algebraic equations. A closed structure involving one or more ***goto*** statements and a set of ***label*** statements is mapped into a set of *statement equations*. The 'solution' of this set of equations is derived using semantics-preserving substitutions. Several key semantics-preserving transformations are also employed (they may be likened to algebraic simplification). Together these substitutions and transformations provide a formal and systematic means for removing ***goto*** statements and, at the same time, optimally restructuring the original delinquent program fragment.

The method offers a rigorous means for re-engineering existing, poorly structured, legacy programs into improved structures that satisfy their original specification. Strongest postcondition calculations may be used to prove that all the transformations employed yield equivalent program components [14–16]. An important feature of the method, apart from its restructuring capability, is that it can detect a significant class of termination problems. The method is easy to apply manually and it is amenable to automation.

2. TRANSFORMING EXITING *GOTO*S INTO INTERNAL *GOTO*S

Two structurally different categories of ***goto*** statements may be encountered. We will first consider ***goto*** statements in loops and subsequently generalize the treatment needed to handle other structural contexts. For loops, the first category is distinguished by the fact that the associated label statement is still within the same loop body. This is called an *internal **goto** statement.* The second category of ***goto*** statement is one that is used to transfer control out of a loop—in this case the label is external to the loop. It is called an *exiting **goto** statement.* The concept of internal/exiting ***goto***s is not limited to loop bodies. It can be extended to model closed blocks. If we can remove both these categories of ***goto*** we can eliminate all ***goto***s from any program structure.

In this section we will deal with removing exiting ***goto*** statements. We assume any exiting ***goto*** statement in a given loop body is *guarded*. If it is not, it must occur in a sequential loop body of the form ***do*** $G \rightarrow S;$ ***goto*** $L;\ S'$ ***od***, where the label L does not appear in S or S'. The control-flow after execution of S will terminate the iteration and S' will never be executed. In this situation, the loop should be replaced by a branch statement ***if*** $G \rightarrow S;$ ***goto*** L ***fi*** and the redundant statement sequence S' should be removed. In the following we always treat any exiting ***goto*** as a guarded exiting ***goto***.

Let us consider a loop with a guarded exiting ***goto***

statement, denoted by ***do*** $G \rightarrow S$; ***if*** $C \rightarrow S_1$; ***goto*** L; $S_2 [] \neg C \rightarrow S_3$ ***fi***; S' ***od***. The function of this ***goto*** is to transfer control out of the loop. It can be replaced by a ***break*** statement which terminates the loop (note a ***break***, which is semantically equivalent to a specialized forward-only ***goto***, is much easier to remove—see below). After termination the control-flow needs to be transferred to the statement labelled L. This suggests implementation by a statement ***goto*** L placed immediately following the loop. The problem with this is that if we use the desired transformation ***do*** $G \rightarrow S$; ***if*** $C \rightarrow S_1$; ***break*** $[] \neg C \rightarrow S_3$; S' ***fi od***; ***goto*** L, the normal exit (at the loop guard) will, on termination, also transfer control to the statement labelled by L. A guard is therefore needed to distinguish the different exits. The simplest way to achieve this is to introduce a fresh *boolean* variable *jump* and initialize it to false. Then, if the ***goto*** L is to be executed the *jump* must be set to true prior to executing the ***break*** that has been introduced into the loop. We then have:

Rule for removal of a guarded exiting ***goto*** (REG):

do $G \rightarrow S$; ***if*** $C \rightarrow S_1$; ***goto*** L $[] \neg C \rightarrow S_3$; ***fi***; S' ***od***
$\models$ *jump* := false;
do $G \rightarrow S$; ***if*** $C \rightarrow S_1$; *jump* := true; ***break***
$[] \neg C \rightarrow S_3$; S' ***fi*** *od*;
if *jump* $\rightarrow$ ***goto*** L ***fi***

where *jump* is a fresh variable

All exiting ***goto***s can be removed from a loop by use of ***break***s and fresh variables. For example, consider a loop with a nested subloop which contains a guarded exiting ***goto*** statement:

do $G \rightarrow S$; ***do*** $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$;
goto L; S_2 $[] \neg C \rightarrow S_3$ ***fi***; S'' ***od***; S''' ***od***

After application of REG twice, we can convert the exiting ***goto*** statement into an internal ***goto*** statement, that is:

Therefore, to remove all exiting ***goto***s from a loop body we first employ the REG transformation as many times as necessary. We start the process by removing ***goto***s from internal nested subloops. The newly produced guarded ***goto***s then become guarded ***goto***s for the external loops. Repeating this process, all exiting ***goto***s including the newly introduced ***goto***s will finally be removed from the transformed loop structure. All occurrences of exiting ***goto***s will then be replaced by ***break***s with all exiting ***goto***s assuming the status of internal ***goto***s. Subsequently the process of removing the flags that have been introduced and the ***break***s can be accomplished by loop rationalization which is discussed in the companion paper [17]. Finally, the newly produced internal ***goto***s can be removed by the method we will describe in the rest of this paper.

3. THEORETICAL BASIS FOR REMOVING *GOTO* STATEMENTS

In this section we introduce a formal method that is suitable for eliminating ***goto***s from any program structure. Since the elimination of exiting ***goto***s has been dealt with, the remaining task is to handle the elimination of ***goto***s from a sequence. This corresponds to elimination of internal ***goto***s from a given closed block (including a loop body).

Initially we will assume all ***goto***s to be processed are not exiting to a guarded structure. A statement ***goto*** L is called *a* **goto** *statement exiting to a guarded structure* if the label L occurs in a guarded statement, such as ***if*** $C \rightarrow \ldots$ L: $\ldots [] \neg C \rightarrow \ldots$ ***fi*** or ***do*** $G \rightarrow \ldots$ L: $\ldots$ ***od***. The case of ***goto***s exiting to a guarded structure will be dealt with later in section 3.4. All labels corresponding to internal ***goto***s here will therefore occur in a sequential structure.

3.1. Statement variables and statement variable equations

To reason systematically and formally about structures that contain internal ***goto***s it is necessary to introduce some formal structures. We borrow algebraic concepts of equation and solution to describe this process. *Statement variable equations* play the most important role. We will show how to construct these equations from program code,

do $G \rightarrow S$; ***do*** $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$; ***goto*** L; $S_2 [] \neg C \rightarrow S_3$ ***fi***; S'' ***od***; S''' ***od***
$\models$ ***do*** $G \rightarrow S$;
jump$'$:= false;
do $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$; *jump*$'$:= **true**; ***break*** $[] \neg C \rightarrow S_3$ ***fi***; S'' ***od***;
if *jump*$'$ $\rightarrow$ ***goto*** L ***fi***;
S'''
od (applying REG for the subloop)
$\models$ *jump* := false;
do $G \rightarrow S$;
jump$'$:= false;
do $G' \rightarrow S'$; ***if*** $C \rightarrow S_1$; *jump*$'$:= **true**; ***break*** $[] \neg C \rightarrow S_3$ ***fi***; S'' ***od***;
if *jump*$'$ $\rightarrow$ *jump* := **true**; ***break*** $[] \neg$ *jump*$'$ $\rightarrow S'''$ ***fi***
od;
if *jump* $\rightarrow$ ***goto*** L ***fi*** (applying REG for the outer loop)

and how to reason with them to generate fruitful restructured results.

Consider a sequence containing n labels, denoted by S_0; ***label***$_1$: S_1; ***label***$_2$: S_2; ...; ***label***$_n$: S_n. We may introduce n *statement variables* [14] X_i where $i \in [1, n]$. X_i denotes a statement sequence commencing from the statement S_i and extending to the end of the sequence. This means $X_i = S_i$; ***label***$_{i+1}$: S_{i+1}; ...; ***label***$_n$: S_n for $i \in [1, n-1]$ and $X_n = S_n$. Schematically shown in Figure 1.

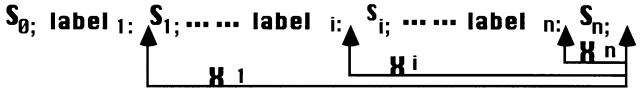


FIGURE 1.

From these relations it follows that $X_i = S_i$; X_{i+1} for $i \in [1, n-1]$ and $X_n = S_n$. Since the block that is executed after execution of ***goto label***$_j$ is X_j, where $j \in [1, n]$, we can build a *statement variable equation* set [18] consisting of n equations of the form:

$$X_i = (S_i;\ \textbf{\textit{label}}_{i+1}:\ S_{i+1})\ [|\textbf{\textit{label}}_1/X_1, \textbf{\textit{label}}_2/X_2, \ldots, \textbf{\textit{label}}_n/X_n|] \quad \text{for any } i \in [1, n-1]$$

$$\text{and } X_n = S_n[|\textbf{\textit{label}}_1/X_1, \textbf{\textit{label}}_2/X_2, \ldots, \textbf{\textit{label}}_n/X_n|]$$

where the notation $S[|\textbf{\textit{label}}_1/X_1, \textbf{\textit{label}}_2/X_2, \ldots, \textbf{\textit{label}}_n/X_n|]$ means substitution of the earliest occurring unguarded sequence that starts with ***goto label***$_i$; ... or ***label***$_i$: S_i; ... by X_i for each branch of S. Note S_i may contain **gotos** but no labels. To understand this substitution, let us consider the three possible situations:

- where S_i does not contain any ***goto***, then $X_i = S_i$; X_{i+1} or $X_n = S_n$;
- where S_i contains an unguarded ***goto label***$_j$, i.e., $S_i = SS_i$; ***goto label***$_j$; SS_{i+1}, then

  $X_i = SS_i; X_j$ where SS_i does not involve any ***goto***s;
- otherwise S_i may always be represented, in general, by

  SS_0; ***if*** $C_1 \rightarrow SS_1$ $[]\, C_2 \rightarrow SS_2\, [] \ldots []\, C_m \rightarrow SS_m$ ***fi***; SS_{m+1}

  where SS_0 does not involve any ***goto***s, and $\forall i \forall j\ ((i, j \in [1, m] \wedge j \neq i) \Rightarrow (C_i \Rightarrow \neg C_j))$ and $C_1 \vee C_2 \vee \ldots \vee C_m \equiv$ true, then the equation is recursively defined as:

$$\begin{aligned} X_i = SS_0;\ \textbf{\textit{if}}\ & C_1 \rightarrow (SS_1; SS_{m+1}; \textbf{\textit{label}}_{i+1}:\ S_{i+1}) \\ & [|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|] \\ & []\, C_2 \rightarrow (SS_2; SS_{m+1}; \textbf{\textit{label}}_{i+1}:\ S_{i+1}) \\ & [|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|] \\ & [] \ldots \\ & []\, C_m \rightarrow (SS_m; SS_{m+1}; \textbf{\textit{label}}_{i+1}:\ S_{i+1}) \\ & [|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|] \\ & \textbf{\textit{fi}}. \end{aligned}$$

These definitions precisely capture the semantics of statement variables. The equation describes recursive relations among the n statement variables. It also describes the control-flow behaviour during execution of this sequence. For example, the equation $X_i = SS_i$; ***if*** $C \rightarrow SS_j$; X_j ***fi*** indicates that after execution of SS_i the control-flow checks the condition C. If it holds, then SS_j executes followed by the statements corresponding to X_j. If C does not hold, termination will take place. An equation of the form $X = S$; ***if*** $C \rightarrow S'; X$ ***fi*** indicates a loop structure for X. According to the semantics it may therefore be replaced by $X = S$; ***do*** $C \rightarrow S'$; S ***od***.

When a statement variable equation is of the form:

$$\begin{aligned} X_i = SS_0;\ \textbf{\textit{if}}\ & C_1 \rightarrow SS_1;\ X_{i_1}\, [] \ldots []\, C_{t-1} \rightarrow SS_{t-1}; X_{i_{t-1}}; \\ & []\, C_t \rightarrow SS_t\, [] \ldots []\, C_m \rightarrow SS_m\ \textbf{\textit{fi}} \end{aligned}$$

where SS_j does not involve any statement variables or ***goto***s, $j \in [1, m]$ and $m \geqslant 0$,

we call it a *standard equation*. It encompasses the simplified forms $X_i = SS_0$; SS_p, $X_i = SS_0$; SS_q; X_q and $X_i = SS_0$; ***if*** $C_q \rightarrow SS_q$; X_q ***fi***, where $p \in [t, m]$ and $q \in [1, t-1]$.

3.2. Sequentializing nested selections

Some transformations are needed to convert statement variable equations directly from raw code into standard equations that are easier to apply substitution to and to reason about. Our approach is similar to the strategy used to solve individual and sets of algebraic equations. For example, we need to transform the equation $x = a^*(b - c^*(x + d))$ into the form $x = a^*b - a^*c^*x - a^*c^*d$ as a preliminary step to solving it for x, i.e. $x = (a^*b - a^*c^*d)/(1 + a^*c)$.

The notation $S[|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|]$ has a number of obvious properties. First, each terminal branch in $S[|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|]$ either contains no statement variable or it ends at a single statement variable. Secondly, in each branch of $S[|\textbf{\textit{label}}_1/X_1, \ldots, \textbf{\textit{label}}_n/X_n|]$, if a statement variable is guarded, it may be guarded by a number of branch statements. The following example illustrates this:

if $C \rightarrow S;$
  if $C' \rightarrow S_1;\ X_1$
  $[]\neg C' \rightarrow S_2$
  fi
$[]\neg C \rightarrow S_3;\ X_3$
fi

To handle structures like this we must first promote all statement variables in the nested branches to their highest possible level in the statement variable equation and then form a standard equation. This requirement can be described

as a transformation from a statement of the form:

$$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C_2 \rightarrow S_2\\ \qquad\qquad [\,]\neg C_2 \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ [\,]\neg C_1 \rightarrow S_4;\\ \qquad\qquad \textbf{\textit{if}}\ C_3 \rightarrow S_5\\ \qquad\qquad [\,]\neg C_3 \rightarrow S_6\\ \qquad\qquad \textbf{\textit{fi}}\\ \textbf{\textit{fi}}\end{array}$$

into an equivalent statement that has one or the other of the following forms:

$$\begin{array}{l}\textbf{\textit{if}}\ C_1' \rightarrow S_1;\ S_2\\ [\,]\,C_2' \rightarrow S_1;\ S_3\\ [\,]\,C_3' \rightarrow S_4;\ S_5\\ [\,]\,C_4' \rightarrow S_4;\ S_6\\ \textbf{\textit{fi}}\end{array}\quad \text{or} \quad \begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1\\ [\,]\,\neg C_1 \rightarrow S_4\\ \textbf{\textit{fi}};\\ \textbf{\textit{if}}\ C_1' \rightarrow S_2\\ [\,]\,C_2' \rightarrow S_3\\ [\,]\,C_3' \rightarrow S_5\\ [\,]\,C_4' \rightarrow S_6\\ \textbf{\textit{fi}}\end{array}$$

Several manipulations allow us to complete these conversions. One of these equivalence transformation rules [16] is:

- $$\begin{array}{l}\textbf{\textit{if}}\ C \rightarrow S_1;\ S\\ [\,]\neg C \rightarrow S_2;\ S\\ \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\textbf{\textit{if}}\ C \rightarrow S_1\\ [\,]\neg C \rightarrow S_2\\ \textbf{\textit{fi}};\\ S\end{array}$$

Another transformation is:

- $$\begin{array}{l}\{P\}\ \textbf{\textit{if}}\ C \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C' \rightarrow S_2\\ \qquad\qquad [\,]\neg C' \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ \qquad [\,]\neg C \rightarrow S_4;\ S_2\\ \qquad \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\{P\}\ \textbf{\textit{if}}\ C \rightarrow S_1\\ \qquad [\,]\neg C \rightarrow S_4\\ \qquad \textbf{\textit{fi}};\\ \qquad \textbf{\textit{if}}\ C' \rightarrow S_2\\ \qquad [\,]\neg C' \rightarrow S_3\\ \qquad \textbf{\textit{fi}}\end{array}$$

where the *strongest postcondition* calculation yields $sp(P \wedge \neg C, S_4) \Rightarrow C'$, for the precondition P.

The *weakest precondition* calculations [16] yield:

- $$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C_2 \rightarrow S_2\\ \qquad\qquad [\,]\neg C_2 \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ \qquad [\,]\neg C_1 \rightarrow S_4;\\ \qquad\qquad\qquad \textbf{\textit{if}}\ C_3 \rightarrow S_5\\ \qquad\qquad\qquad [\,]\neg C_3 \rightarrow S_6\\ \qquad\qquad\qquad \textbf{\textit{fi}}\\ \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\textbf{\textit{if}}\ C_1 \wedge wp(S_1, C_2) \rightarrow S_1; S_2\\ [\,]\,C_1 \wedge \neg wp(S_1, C_2) \rightarrow S_1; S_3\\ [\,]\neg C_1 \wedge wp(S_4, C_3) \rightarrow S_4; S_5\\ [\,]\neg C_1 \wedge \neg wp(S_4, C_3) \rightarrow S_4; S_6\\ \textbf{\textit{fi}}\end{array}$$

As part of these transformations it is sometimes necessary to move an assignment or some other statement inside a guarded statement. In our companion paper on loop reengineering [17] we describe how to do this using strongest postcondition calculations. The following example illustrates the required transformation:

$$\begin{array}{l}x := x + y^*z;\ \textbf{\textit{if}}\ x - y > 0 \rightarrow y := \ldots\\ \qquad\qquad [\,]x - y \leqslant 0 \rightarrow z := \ldots\ \textbf{\textit{fi}}\\ |= \textbf{\textit{if}}\ x + y^*z - y > 0 \rightarrow x := x + y^*z;\ y := \ldots\\ \qquad [\,]x + y^*z - y \leqslant 0 \rightarrow x := x + y^*z;\ z := \ldots\ \textbf{\textit{fi}}\end{array}$$

Referring back to the original form introduced above, when $S_1 = \textbf{\textit{skip}}$, $wp(S_1, C_2) \equiv C_2$ and $wp(S_1, \neg C_2) \equiv \neg C_2$ the transformation rule above becomes:

$$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow \textbf{\textit{if}}\ C_2 \rightarrow S_2\ [\,]\neg C_2 \rightarrow S_3\ \textbf{\textit{fi}}\ [\,]\neg C_1 \rightarrow S_4; \ldots \textbf{\textit{fi}}\\ |= \textbf{\textit{if}}\ C_1 \wedge C_2 \rightarrow S_2\ [\,]\,C_1 \wedge \neg C_2 \rightarrow S_3\ [\,]\neg C_1 \rightarrow S_4; \ldots \textbf{\textit{fi}}\end{array}$$

The weakest preconditions $wp(S_1, C_2)$ or $wp(S_4, C_3)$ may be difficult to calculate (when S_1 or S_4 involve loop structures) or even undefined [when S_1 or S_4 involve a ***read*(x)** and the branch guards C_2 or C_3 also involve the variable **x**] for arbitrary input data. The following rule can, however, always realize the required transformation:

Sequentializing Nested Selection (SNS):

$$\begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\\ \qquad\qquad \textbf{\textit{if}}\ C_2 \rightarrow S_2\\ \qquad\qquad [\,]\neg C_2 \rightarrow S_3\\ \qquad\qquad \textbf{\textit{fi}}\\ [\,]\neg C_1 \rightarrow S_4;\\ \qquad\qquad \textbf{\textit{if}}\ C_3 \rightarrow S_5\\ \qquad\qquad [\,]\neg C_3 \rightarrow S_6\\ \qquad\qquad \textbf{\textit{fi}}\\ \textbf{\textit{fi}}\end{array} \quad |= \quad \begin{array}{l}\textbf{\textit{if}}\ C_1 \rightarrow S_1;\ \textit{flag} := \text{true}\\ \qquad [\,]\neg C_1 \rightarrow S_4;\ \textit{flag} := \text{false}\\ \qquad \textbf{\textit{fi}};\\ \qquad \textbf{\textit{if}}\ C_2 \wedge \textit{flag} \rightarrow S_2\\ \qquad [\,]\neg C_2 \wedge \textit{flag} \rightarrow S_3\\ \qquad [\,]\,C_3 \wedge \neg\, \textit{flag} \rightarrow S_5\\ \qquad [\,]\neg C_3 \wedge \neg\, \textit{flag} \rightarrow S_6\\ \qquad \textbf{\textit{fi}}\end{array}$$

where *flag* is a fresh variable

As for REG, we may remove *flag*s using the technique described in the companion paper [17]. The process of converting any non-standard equation into a standard equation may involve a number of applications of sequentializing nested selections (SNS).

We also require that no statement variable X occurs more than once on the right-side of any standard equation. A disjunctive rule can always be used to eliminate duplicate occurrences. The example shown below illustrates the

transformation:

$$\textbf{\textit{if}}\ C_1 \to S_1; X[]\, C_2 \to S_2; X[]\, C_3 \to S_3[] \ldots []C_p \to S_p\ \textbf{\textit{fi}}$$
$$\models \textbf{\textit{if}}\ C_1 \vee C_2 \to \textbf{\textit{if}}\ C_1 \to S_1[]C_2 \to S_2\ \textbf{\textit{fi}}; X\ []C_3 \to S_3[] \ldots []C_p \to S_p\ \textbf{\textit{fi}}$$

3.3. Non-recursive solutions for standard equations

In the following sections, we will use a graph theoretic form to describe a systematic process for finding solutions for sets of standard equations. This form not only indicates detailed steps for resolution but it also suggests how the whole approach may be implemented.

3.3.1. Variable dependency graph

Consider a set of standard equations:

$$X_i = SS_0; \textbf{\textit{if}}\ C_1 \to SS_1; X_{i_1}[] \ldots []\, C_{t-1} \to SS_{t-1}; X_{i_{t-1}} []\, C_t \to SS_t[] \ldots []\, C_m \to SS_m\ \textbf{\textit{fi}}$$

where $i \in [1,n]$ and $i_j \in [1,m]$ for any $j \in [1,t-1], t \geqslant 1$.

Each equation corresponds to a node in a directed graph. Nodes are connected according to their variable dependency relations, i.e. the directed graph is formally defined as $\langle \{X_1, X_2, \ldots, X_n\}, E\rangle$, where E contains all edges $\langle X_i, X_s\rangle$ such that SS_0; ***if*** $C_1 \to SS_1$; $X_{i_1}[] \ldots []\ C_{t-1} \to SS_{t-1}$; $X_{i_{t-1}}[]\, C_t \to SS_t[] \ldots []\, C_m \to SS_m$ ***fi*** involves X_s, i.e. $X_s \in \{X_{i_1}, X_{i_2}, \ldots X_{i_{t-1}}\}$. The node X_i has $t-1$ children $\{X_{i_1}, X_{i_2}, \ldots X_{i_{t-1}}\}$.

Obviously any leaf node X_{leaf} that is not reflexive (i.e. it does not have a graph theoretic loop edge attached) indicates that the corresponding equation is $X_{leaf} = S_{leaf}$ where S_{leaf} does not involve any statement variable X_i, $i \in [1,n]$. We call such a S_{leaf} the *Solution* of X_{leaf} and the leaf X_{leaf} is said to be coloured. Generally any node is said to be coloured if we have obtained an equivalent ***goto***-free solution for its corresponding statement variables. All non-cycle leaves in the directed graph may be treated as coloured.

When all the children $\{X_{i_1}, X_{i_2}, \ldots X_{i_{t-1}}\}$ of a node X_i are coloured, it too can be coloured. The colouring of X_i maps in the equation domain to a substitution that will yield a solution for X_i. The substitution rule is described below.

Rule of Substitution Solution (RSS):

Given any (standard) statement equation

$$X_i = SS_0; \textbf{\textit{if}}\ C_1 \to SS_1; X_{i_1}[] \ldots []C_{t-1} \to SS_{t-1}; X_{i_{t-1}} []C_t \to SS_t[] \ldots []C_m \to SS_m\ \textbf{\textit{fi}}$$

and all the solutions Y_{ij} to X_i's children such that $X_{i_j} = Y_{i_j}$, $j \in [1,t-1]$, then the segment

$$SS_0; \textbf{\textit{if}}\ C_1 \to SS_1; Y_{i_1}[] \ldots []\, C_{t-1} \to SS_{t-1}; Y_{i_{t-1}} []\, C_t \to SS_t[] \ldots []\, C_m \to SS_m\ \textbf{\textit{fi}}$$

is the *Solution* for X_i.

The rule of substitution solution (RSS) suggests that we can easily obtain solutions for all statement variables. A sequence of substitutions map their statement equations to a tree(s) (without any cycle-nodes). This tree-colouring process corresponds to an extended *Post-Order* traversal, first colouring the leaves, then proceeding progressively up the tree to the root. Obviously, these solutions are ***goto***-free code segments that are semantically equivalent to their corresponding statement variables. Each colouring step defines a semantically equivalent ***goto***-free solution for a node.

The RSS rule also provides a post-order colouring strategy for any directed graph that is based on statement equations. It follows that if we can also colour cycles, we can then colour any directed graph, and obtain equivalent ***goto***-free solutions for the statement equations.

3.3.2. Self-cycle removal (LE)

Any cycle is either a self-cycle involving a single node or a complex cycle that involves more than one node. Given any self-cycle node X, its corresponding statement equation is, in general, of the form $X = S$; ***if*** $C \to S'$; $X\,[]\neg C \to Y$ ***fi***. Its semantics indicates that after execution of S the control-flow iteratively executes S'; S until the condition C fails then Y is executed. We therefore have the following colouring rule for LE:

Loop Extraction (LE):
Given any (standard) statement equation $X = S$; ***if*** $C \to S'; X\,[]\neg C \to Y$ ***fi***, the statement S; ***do*** $C \to S'$; S ***od***; Y is the *Solution* for X.

The equation $X = S; S'; X$, (corresponding to $C = \text{true}$ in LE) has a non-terminating solution S; ***do*** true $\to S'$; S ***od*** (or ***do*** true $\to$ S; S' ***od***). This form can be used to detect non-termination defects.

Using SNS, RSS and LE we can always complete the task of finding a solution for any directed graph that contains only self-cycles. From a graph theoretic standpoint, the rule LE absorbs any self-cycle node into a super-node. Therefore application of LE yields a new directed graph that is a tree(s) without any cycles. The latter can be coloured using RSS.

3.3.3. Cycle colouring

Before considering complex cycle colouring we need to review the rules RSS and LE. Given any standard equation $X = S$; ***if*** $C \to S'$; $X_a\,[]\neg C \to S''$; X_b ***fi***, RSS indicates that the substitution for X_a and X_b by their solutions Y_a and Y_b yields an equivalent equation $X = S$; ***if*** $C \to S'$; $Y_a\,[]\neg C \to S''$; Y_b ***fi*** (at this time S; ***if*** $C \to S'$; $Y_a\,[]\neg C \to S''$; Y_b ***fi*** contains no ***goto***s). Furthermore, if we substitute for X_a and X_b using the equations $X_a = S_a$; ***if*** $C_a \to S_a'$; $X_{a1}\,[]\neg C_a \to S_a''$; X_{a2} ***fi*** and $X_b = S_b$; ***if*** $C_b \to S_b'$; $X_{b1}\,[]\neg C_b \to S_b''$; X_{b2} ***fi***, the resulting substitutions should still maintain the original

equation, i.e.,

$$
\begin{aligned}
X = S;\ \textbf{\textit{if}}\ C \rightarrow S';\ S_a;\\
&\textbf{\textit{if}}\ C_a \rightarrow S'_a;\ X_{a1}\\
&[\,]\neg C_a \rightarrow S''_a;\ X_{a2}\\
&\textbf{\textit{fi}}\\
[\,]\neg C \rightarrow S'';\ S_b;\\
&\textbf{\textit{if}}\ C_b \rightarrow S'_b;\ X_{b1}\\
&[\,]\neg C_b \rightarrow S''_b;\ X_{b2}\\
&\textbf{\textit{fi}}.\\
\textbf{\textit{fi}}
\end{aligned}
$$

After sequentializing the nested selection, i.e. standardizing the substituted equations, we obtain a new standard equation for X which involves the statement variables X_{a1}, X_{a2}, X_{b1} and X_{b2}.

The role of RSS is to convert the statement-variable-dependent relation of a standard equation. That is, after applying RSS and SNS, the equation for X should only depend on the statement variables which are the child nodes (i.e., X_{a1}, X_{a2}, X_{b1} and X_{b2}) of X's direct child nodes (X_a and X_b). Similarly, the LE rule also applies for equations in the form $X = S;\ \textbf{\textit{if}}\ C \rightarrow S';\ X\,[\,]\neg C \rightarrow X'\ \textbf{\textit{fi}}$, where X' is a statement variable. The solution is $X = S;\ \textbf{\textit{do}}\ C \rightarrow S';\ S\ \textbf{\textit{od}}\ ;\ X'$. In summary, RSS together with SNS and LE allow us to transform by substitution a set of standard equations in a similar manner to the way we solve a set of algebraic equations.

Now let us consider a complex cycle with more than one node. We select a node X_1 as a *virtually coloured* node. This means that we treat the statement variable X_1 as a 'solution' during the following *virtual colouring* process. This step corresponds to converting the cycle into a tree structure, where all direct parents of the original node X_1 become the direct parents of the leaf X_1 (corresponding to the virtually coloured node) and the original node X_1 also becomes a root of the tree structure (see Figure 2).

When this resulting tree structure contains no complex cycles, all nodes can be coloured by LE and RSS. However, their solutions still contain the statement variable X_1. For this reason we call their solutions *virtual solutions* and this colouring process a *virtual colouring*. The third step is to actually colour the node X_1. Because the root's equation, after virtual colouring, contains X_1 only, we can use LE to obtain an actual solution for X_1. After substituting the actual solution for X_1 for all virtual solutions, we obtain the actual solutions for all other nodes on the cycle. When the resulting tree structure contains other complex sub-cycles, the first step (selecting a virtually coloured node) and the second step (virtually colouring) may need to be applied recursively. The third step (actually colouring) then needs to proceed progressively up the tree to the root X_1. This process is similar to the process of using the algebraic resolution method to find a solution for a set of algebraic equations $X_i = f_i(X_1, X_2, \ldots, X_m), i \in [1, m]$.

To illustrate the process of complex cycle colouring let us consider a three-node complete directed graph as an example (see Figure 3). We use a set of arbitrary equations $X_i = f_i(X_1, X_2, X_3), i \in [1, 3]$, as an example to show the process of colouring a complex cycle (see Table 1).

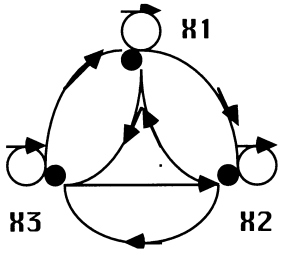


FIGURE 3.

In general, cycle-colouring (for more than one node) employs the following process:

Step 1: take a node as a virtually coloured node;
Step 2: apply the tree-colouring process to colour other nodes in the cycle until all nodes are (virtually) coloured. To achieve this, Step1, Step2 may need to be applied recursively;
Step 3: (actually) colour the virtually coloured nodes then actually colour all other nodes in the cycle.

Intuitively the process of complex cycle-colouring involves transforming and converting various forms of statement equation into standard ones using SNS, RSS and LE. We then apply LE (to find a real solution) as the last substitution (actually colouring all nodes). A number of strategies have been developed to improve the colouring process. Full details are provided in a separate report [18].

3.4. Removing *GOTO* statements exiting to guarded structures

We can now deal with ***goto*** statements exiting to guarded structures. For these cases, as long as we can correctly build statement variable equations, the method we have

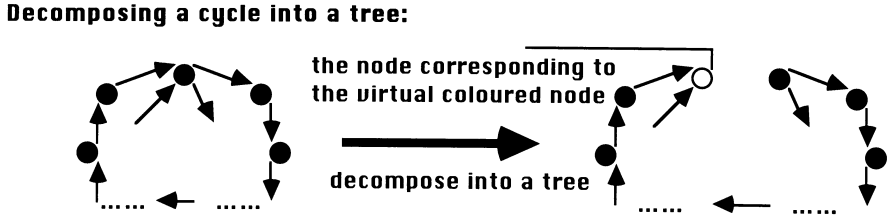


FIGURE 2.

TABLE 1

No.	*Input form*	*Output form*	*Comment*
1	$X_1 = f_1(X_1, X_2, X_3)$ $X_2 = f_2(X_1, X_2, X_3)$ $X_3 = f_3(X_1, X_2, X_3)$	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$	Use the rule LE to remove self-cycles and transform the input equations
2	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$		Select X_1 as a virtually coloured node to colour the graph, but we meet the sub-cycle (X_2, X_3)
3	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$		Repeat virtual colouring process, by selecting X_2 node as another virtually coloured node. Now $X_3 = g_3(X_1, X_2)$ is a virtually coloured solution
4	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, X_3)$ $X_3 = g_3(X_1, X_2)$	$X_2 = g_2(X_1, g_3(X_1, X_2))$	Use the rule RSS to virtually colour the node X_2
5	$X_1 = g_1(X_2, X_3)$ $X_2 = g_2(X_1, g_3(\ldots))$ $X_3 = g_3(X_1, X_2)$	$X_2 = h_2(X_1)$	Use the rule LE to obtain a virtually coloured solution
6	$X_1 = g_1(X_2, X_3)$ $X_2 = h_2(X_1)$ $X_3 = g_3(X_1, X_2)$	$X_3 = h_3(X_1)$	Then colour the node X_3 by the rule RSS: $X_3 = g_3(X_1, h_2(X_1)) = h_3(X_1)$
7	$X_1 = g_1(X_2, X_3)$ $X_2 = h_2(X_1)$ $X_3 = h_3(X_1)$	$X_1 = g_1(h_2(X_1), h_3(X_1))$	From the virtually coloured results of X_2 and X_3, actually colour X_1 by the rule RSS
8	$X_1 = g_1(\ldots, \ldots)$ $X_2 = h_2(X_1)$ $X_3 = h_3(X_1)$	$X_1 = Y_1$	Apply the rule LE for the equation: $X_1 = g_1(h_2(X_1), h_3(X_1))$
9	$X_1 = Y_1$ $X_2 = h_2(X_1)$ $X_3 = h_3(X_1)$	$X_2 = Y_2$ $X_3 = Y_3$	Apply the rule RSS for the equations $X_2 = h_2(X_1)$ and $X_3 = h_3(X_1)$ to actually colour X_2 and X_3

described is still applicable to remove this form of ***goto*** statement. Provided we can identify the statements that follow a label within a guarded statement, then the statement equation can be easily built. For any branch statement

$$\textbf{\textit{if}}\ C \rightarrow S;\ \textbf{\textit{label}}: S' \,[\,]\neg C \rightarrow S''\ \textbf{\textit{fi}};\ S'''$$

the statement sequence that follows the ***label*** is certainly $S'; S'''$. We can build its corresponding statement equation $X = (S'; S''')\,[|\textbf{\textit{label}}/X, \ldots |]$. For instance, the labels ***label2***, ***label3*** and ***label4*** in Fermat's Algorithm below are guarded. We have used this principle to form their equations.

For those cases where a label occurs in a sequential loop body, i.e.

$$\textbf{\textit{do}}\ G \rightarrow S;\ \textbf{\textit{label}}: S'\ \textbf{\textit{od}};\ S''$$

the statement that follows the label ***label*** is S'; ***do*** $G \rightarrow S$; ***label***$: S'$ ***od***; S''. We can construct its corresponding statement equation as $X = (S';$ ***if*** $G \rightarrow S;$ $X\,[\,]\neg G \rightarrow S''$ ***fi***$)\,[|\textbf{\textit{label}}/X, \ldots |]$ because $S';$ ***do*** $G \rightarrow S;$ ***label***$: S'$ ***od***; S'' is equivalent to $S';$ ***if*** $G \rightarrow S;\ X\,[\,]\neg G \rightarrow S''$ ***fi*** since X represents the statement that follows the label ***label*** to the end. Alternatively we may use the equation $X = S';$ ***do*** $G \rightarrow S;\ S'$ ***od***; $S''[|\textbf{\textit{label}}/X, \ldots |]$. This is just the result obtained after applying LE to the original equation.

Using either of these methods we can always define a statement variable equation for a label which is guarded by complex guarded statements.

For example, with the label in

$$\textbf{\textit{do}}\ G \rightarrow S;\ \textbf{\textit{if}}\ C \rightarrow S_1;\ \textbf{\textit{label}}: S_2\,[\,]\neg C \rightarrow S_3\ \textbf{\textit{fi}};\ S_4\ \textbf{\textit{od}};\ S_5$$

its corresponding statement equation is

$$X = S_2;\ S_4;\ \textbf{\textit{do}}\ G \rightarrow S;\ \textbf{\textit{if}}\ C \rightarrow S_1;\ S_2\,[\,]\neg C \rightarrow S_3\ \textbf{\textit{fi}};\ S_4\ \textbf{\textit{od}};\ S_5[|\textbf{\textit{label}}/X, \ldots |]$$

We can always use this method to remove any internal ***goto*** statements from a loop body or any sequential statement block.

In this section we have presented a strategy for removing ***goto*** statements from any program. It may introduce a number of ***break***s. However, a previously developed process [17] enables us to eliminate these ***break***s and produce an equivalent, transformed program, that is ***goto***-free and ***break***-free.

4. USING THE STATEMENT EQUATION SOLUTIONS TO RECONSTRUCT PROGRAMS

For any equation set containing n statement equations we assume each solution to be Y_i, where $i \in [1, n]$. The solution Y_i gives us the required transformed program segment. The remaining task is to correctly substitute the solution. Since the solution Y_i represents a semantically

equivalent statement sequence starting from the statement labelled by ***label***$_i$ and proceeding to the end, then the following rule may be applied:

Program Reconstruction Rule (PRR)

Given any program/segment S_0; ***label***$_1$: S_1; ***label***$_2$: S_2; ...; ***label***$_n$: S_n where Y_i, $i \in [1, n]$ is the solution to the ith equation, then the equivalent ***goto***-free program/segment is $(S_0$; ***label***$_1$: $S_1)[|$***label***$_1/Y_1$, ***label***$_2/Y_2, \ldots,$ ***label***$_n/Y_n|]$

5. APPLICATION OF THE METHOD

To illustrate the reengineering process we have defined we will apply it to two examples.

Example 1

The first example we will consider involves transforming an *incorrect* implementation of *Fermat's Algorithm* [1] that employs ***goto***s. The original implementation has the form:

$x := 2^*\sqrt{n} + 1$; $y := 1$; $r := (\sqrt{n})^2 - n$;

label *1*: ***if*** $r < 0 \rightarrow$ ***goto label*** *3* ***fi***;

label *2*: $r := r - y$; $y := y + 2$; ***goto label*** *1*;

label *3*: ***if*** $r = 0 \rightarrow$ ***goto label*** *4* ***fi***;
$r := r + x$; $x := x + 2$; ***goto label*** *2*;

label *4*: $u := (x - y)$ div 2

Before building the statement equations, we need to convert the program into a form that corresponds to the standard equations:

label *1*: ***if*** $r < 0 \rightarrow$ ***goto label*** *3*
$[]\, r \geqslant 0 \rightarrow$ ***label*** *2*: $r := r - y$; $y := + 2$; ***goto label*** *1*;
label *3*: ***if*** $r = 0 \rightarrow$ ***goto label*** *4*
$[]\, r \neq 0 \rightarrow r := r + x$;
$x := x + 2$; ***goto label*** *2*;
label *4*: $u := (x - y)$ div 2
fi
fi;

We then can easily build the following statement equation set (see Figure 4).

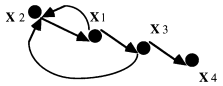


FIGURE 4.

$X_1 =$ ***if*** $r < 0 \rightarrow X_3\, []\, r \geqslant 0 \rightarrow X_2$ ***fi***

$X_2 = r := r - y$; $y := y + 2$; X_1

$X_3 =$ ***if*** $r = 0 \rightarrow X_4 []\, r \neq 0 \rightarrow r := r + x$; $x := x + 2$; X_2 ***fi***

$X_4 = u := (x - y)$ div 2

where $u := (x - y)$ div 2 is already the solution to X_4.

Now we use the colouring algorithm to find solutions. First, we select X_1 as a virtually coloured node. X_2 is then virtually coloured automatically, as is X_3. This process corresponds to the following transformations:

$X_1 =$ ***if*** $r < 0 \rightarrow X_3\, []\, r \geqslant 0 \rightarrow X_2$ ***fi***

$=$ ***if*** $r < 0 \rightarrow$ ***if*** $r = 0 \rightarrow X_4\, []\, r \neq 0 \rightarrow r := r + x$; $x := x + 2$; X_2 ***fi***
$[]\, r \geqslant 0 \rightarrow X_2$
fi (RSS)

$=$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$; $X_2\, []\, r \geqslant 0 \rightarrow X_2$ ***fi*** (since the $r = 0$ branch is unreachable)

$=$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; X_2 (SNS)

$=$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$; X_1 (RSS)

Therefore we actually colour X_1 by a solution (corresponding to a non-terminating loop) ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$ ***od*** according to LE. The solutions for all the equations are:

$Y_1 =$ ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$ ***od***

$Y_2 = r := r - y$; $y := y + 2$; ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow r := r + x$; $x := x + 2$ ***fi***; $r := r - y$; $y := y + 2$ ***od***

$Y_3 =$ ***if*** $r = 0 \rightarrow Y_4\, []\, r \neq 0 \rightarrow r := r + x$; $x := x + 2$; $r := r - y$; $y := y + 2$; ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow \ldots$ ***od fi***

$=$ ***if*** $r = 0 \rightarrow u := (x - y)$ div 2
$[]\, r \neq 0 \rightarrow x := x + 2$; $y := y + 2$; $r := r + x - y$; ***do*** true $\rightarrow$ ***if*** $r < 0 \rightarrow \ldots$ ***od***
fi (removing the redundant assignment $r := r + x$)

$Y_4 = u := (x - y)$ div 2

To reconstruct the Fermat's Algorithm above, we just need to substitute Y_1 for the sequence starting with ***label*** *1*: ***if*** $r < 0 \rightarrow \ldots$; ***label*** *4*: $u := (x - y)\,\mathrm{div}\,2$. Hence we end up with the equivalent program:

$x := 2^*\sqrt{n} + 1;\ y := 1;\ r := (\sqrt{n})^2 - n;$

do true $\rightarrow$ ***if*** $r < 0 \rightarrow x := x + 2;\ r := r + x$ ***fi***;

$y := y + 2;\ r := r + x - y$ ***od***

In this example a non-terminating defect is discovered. We should remark that during transformation of the equation X_1, an unreachable path $r = 0 \rightarrow X_4$ is found under its precondition r<0. This indicates the original program *never* reaches the last statement although the syntactic representation of the program contains this statement. The example demonstrates that our method not only eliminates ***goto***s from programs but that it can be used to detect logical defects and to optimize programs.

Example 2:

As another example, consider a Pascal procedure taken from software used in industry (P. Farrow, personal communication).

```
procedure INTI(var TD, DTD, RKO, T, DT,
                   JS, JN, IO, JS4: integer);
  label 1, 2, 3, 4, 5;
  begin
      IO := RKO; JN := 0;
      if IO = 4 then goto 1;
      JS := JS + 1;
      if JS = 3 then JS := 1; if JS = 2 then goto 5;
      DT := DTD;
  3:  TD := TD + DT; T := TD; goto 5;
  1:  JS4 := JS4 + 1; if JS4 = 5 then JS4 := 1;
      if JS4 = 1 then goto 2;
      if JS4 = 3 then goto 4;
      goto 5;
  2:  DT := DTD div 2;
      goto 3;
  4:  TD := TD + DT; DT := 2*DT; T := TD;
  5: end
```

This problem may most conveniently be restructured by treating it as two distinct sub-problems (Figure 5A). The more complicated of the sub-problems begins with the sequence commencing at label 3. We will deal with this sub-problem first, then incorporate it into an overall solution. Before applying our proposed restructuring method, we transform the segment

$\ldots$; ***if*** JS4 = 1 ***then goto*** 2;
if JS4 = 3 ***then goto*** 4; ***goto*** 5; $\ldots$

into the standard form:

$\ldots$; ***if*** JS4 = 1 ***then goto*** 2 ***elsif*** JS4 = 3
then goto 4 ***else goto*** 5; $\ldots$

Mapping the structure into a statement equation set we get:

$X_1 =$ JS4 := JS4 + 1; ***if*** JS4 = 5 ***then*** JS4 := 1;
if JS4 = 1 ***then*** X_2 ***elsif*** JS4 = 3 ***then*** X_4 ***else*** X_5
$X_2 =$ DT := DTD div 2; X_3
$X_3 =$ TD := TD + DT; T := TD; X_5
$X_4 =$ TD := TD + DT; DT := 2^*DT; T := TD; X_5
$X_5 =$ ***skip***

where X_5 has been coloured already (Figure 5B).

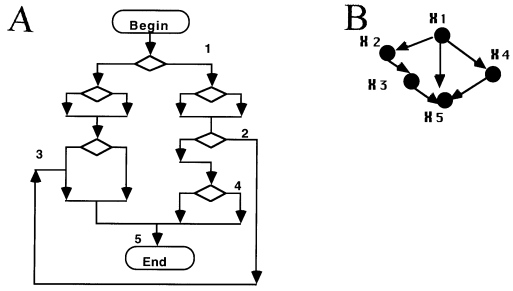


FIGURE 5

The solutions Y_2, Y_3 and Y_4 for X_2, X_3 and X_4 are straightforward to find because there are no cycles involved. We will therefore only show the workings for finding the solution Y_1 for X_1:

$Y_1 =$ JS4 := JS4 + 1; ***if*** JS4 = 5 ***then*** JS4 := 1; ***if*** JS4 = 1 ***then*** Y_2 ***elsif*** JS4 = 3 ***then*** Y_4 ***else*** Y_5

= ***if*** JS4 = 4 ***then*** JS4 := JS4 + 1; JS4 := 1 ***else*** JS4 := JS4 + 1; (SNS)
if JS4 = 1 ***then*** DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 3 ***then*** TD := TD + DT; DT := 2^*DT; T := TD;
else skip (RSS)

= ***if*** JS4 = 4 ***then*** JS4 := 1; ***if*** JS4 = 1 ***then*** DT ... ***elsif*** JS4 = 3 ***then*** TD ... ***else skip***
else JS4 := JS4 + 1; ***if*** JS4 = 1 ***then*** DT ... ***elsif*** JS4 = 3 ***then*** TD ... ***else skip*** (SNS)

= ***if*** JS4 = 4 ***then*** JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 0 ***then*** JS4 := JS4 + 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 2 ***then*** JS4 := JS4 + 1; TD := TD + DT; DT := 2^*DT; T := TD
else JS4 := JS4 + 1 (removing redundancy, SNS)

The solutions for this equation set are therefore:

Y_1 = ***if*** JS4 = 4 ***then*** JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 0 ***then*** JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
elsif JS4 = 2 ***then*** JS4 := 3; TD := TD + DT; DT := 2^*DT; T := TD
else JS4 := JS4 + 1

Y_2 = DT := DTD div 2; TD := TD + DT; T := TD

Y_3 = TD := TD + DT; T := TD

Y_4 = TD := TD + DT; DT := 2^*DT; T := TD

Y_5 = ***skip***

Before substitution of the solutions for this program, we have to transform the original code:

if IO = 4 ***then goto*** 1; JS := JS + 1;
if JS = 3 ***then*** JS := 1;
if JS = 2 ***then goto*** 5; DT := DTD;
3: TD; = TD + DT; . . . . . .

into the equivalent standard form using SNS:

if IO = 4 ***then goto*** 1;
if JS = 2 ***then*** JS := JS + 1; JS := 1;
DT := DTD; (***label***) 3: . . .
elsif JS = 1 ***then*** JS := JS + 1; JS := 2; ***goto*** 5
else JS := JS + 1; DT := DTD; (***label***) 3: . . . ,

and furthermore,

if IO = 4 ***then goto*** 1
elsif JS = 2 ***then*** JS := 1; DT := DTD; (***label***) 3: . . .
(redundant JS := JS + 1 is removed)
elsif JS = 1 ***then*** JS := 2; ***goto*** 5
(redundant JS := JS + 1 is removed)
else JS := JS + 1; DT := DTD; (***label***) 3: . . . ,

We therefore have an equivalent procedure body

IO := RKO; JN := 0;
if IO = 4 → ***if*** JS4 = 4 ***then*** JS4 := 1;
DT := DTD div 2;
TD := TD + DT; T := TD
elsif JS4 = 0 ***then*** JS4 := 1;
DT := DTD div 2;
TD := TD + DT; T := TD
elsif JS4 = 2 ***then*** JS4 := 3;
TD := TD + DT; DT := 2^*DT; T := TD
else JS4 := JS4 + 1
[]IO ≠ 4 ∧ JS = 2 → JS := 1; DT := DTD;
TD := TD + DT; T := TD
[]IO ≠ 4 ∧ JS = 1 → JS := 2
[]IO ≠ 4 ∧ JS ≠ 1 ∧ JS ≠ 2 → DT := DTD;
TD := TD + DT; T := TD
fi

So we end up with the equivalent procedure:

procedure INTI(***var*** TD, DTD, RKO, T, DT, JS, JN, IO, JS4: ***integer***);
begin
IO := RKO; JN := 0;
if IO = 4 ∧ (JS4 = 0 ∨ JS4 = 4) → JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD
[]IO = 4 ∧ JS4 = 2 → JS4 := 3; TD := TD + DT; DT := 2^*DT; T := TD
[]IO = 4 ∧ JS4 ≠ 0 ∧ JS4 ≠ 2 ∧ JS4 ≠ 4 → JS4 := JS4 + 1
[]IO ≠ 4 ∧ JS = 1 → JS := 2
[]IO ≠ 4 ∧ JS = 2 → JS := 1; DT := DTD; TD := TD + DT; T := TD
[]IO ≠ 4 ∧ JS ≠ 1 ∧ JS ≠ 2 → JS := JS + 1; DT := DTD; TD := TD + DT; T := TD
fi
end

which can be easily transformed into the following procedure in Pascal:

```
procedure INTI(var TD, DTD, RKO, T, DT, JS, JN, IO, JS4: integer);
begin
    IO := RKO; JN := 0;
    if IO = 4 then if JS4 = 0 ∨ JS4 = 4 then
                        begin JS4 := 1; DT := DTD div 2; TD := TD + DT; T := TD end
                        else if JS4 = 2 then
                            begin JS4 := 3; TD := TD + DT; DT := 2*DT; T := TD end
                        else JS4 := JS4 + 1
    else if JS = 1 then JS := 2
        else if JS = 2 then begin JS := 1; DT := DTD; TD := TD + DT; T := TD end
            else begin JS := JS + 1; DT := DTD; TD := TD + DT; T := TD end
end
```

This method is very useful for re-engineering complex structures containing ***goto***s. The transformed program is more readable, easier to maintain and more reliable. A number of quality defects, such as redundancy, and non-termination can be removed or detected, and other tasks, such as verification and derivation of a specification from programs, may also be achieved.

4. CONCLUSION

We have introduced a process, that enables the elimination of all ***goto*** statements from any program. The advantage of this approach over other alternatives that have been proposed is that it is securely based on formal semantics. To ensure the generality of this process it has been necessary to formulate a general and powerful formally based method to remove ***goto*** statements from any program structure. The processes we have developed can be used to detect and/or remove a number of quality and reliability defects from programs.

This process specifies a set of general, language-independent, widely applicable, formal, but also practical, techniques for improving the quality of programs. It can render difficult code systematically manageable without the usual tedium of pouring over existing complex program structures. The recommended strategy is first to tame such structures using the methods we have suggested and, only then, to proceed to analyse the code for its intent. The method is quite straightforward to apply manually and it has the potential for automated implementation.

The fact that the key process employed is very similar to that used for solving sets of algebraic equations should make the method attractive to a wide audience. An important use of the method is as a preprocessing step for the more general re-engineering processes: loop rationalization and loop normalization [17, 18]. Another interesting application of the method is to use it directly to eliminate recursion in programs. We can do this because any recursion can be replaced by a mechanism involving ***goto*** statements.

REFERENCES

[1] Dromey, R. G. and McGettrick, A. D. (1992) On Specifying Software Quality. *Software Quality J.*, **1**, 43–74.

[2] Ashcroft, E. and Manna, Z. (1972) The Translation of GOTO Programs to WHILE Programs. In *Proc. IFIP Congr. 71*, pp. 250–255, North-Holland, Amsterdam.

[3] Gray, A. S. and Whitty, R. W. (1982) Correspondence of Flowchart Schemata. *Comp. J.*, **25**, 495.

[4] Knuth, D. E. (1974) Structured Programming with GOTO statements. *ACM Computing Surveys*, **6**, 261–301.

[5] Knuth, D. E. and Floyd, R. W. (1971) Notes on Avoiding GOTO Statements. *Inf. Proc. Letters*, **1**, 23–31.

[6] Oulsnam, G. (1982) Unravelling Unstructured Programs. *Comp. J.*, **25**, 379–387.

[7] Williams, M. H. (1976) Generating Structured Flow Diagrams: the Nature of Unstructuredness. *Comp. J.*, **20**, 45–50.

[8] Williams, M. H. (1982) A Comment on the Decomposition of Flowchart Schemata. *Comp. J.*, **25**, 393–396.

[9] Williams, M. H. (1983) Flowchart Schemata and the Problem of Nomenclature. *Comp. J.*, **26**, 270–276.

[10] Williams, M. H. and Chen, G. (1985) Restructuring Pascal Programs Containing GOTO Statements. *Comp. J.*, **28**, 134–137.

[11] Williams, M. H. and Ossher, H. L. (1978) Conversion of Unstructured Flow Diagrams to Structured Form. *Comp. J.*, **21**, 161–167.

[12] Carpenter, B. E., Doran, R. W. and Hopper, K. (1977) Non-recursive Recursion. *Softw. Pract. Exp.*, **7**, 263–268.

[13] Goldschlager, L. M. (1981) Recursion in Small Storage. *Softw. Pract. Exp.* **11**, 745–751.

[14] Back, R. J. R. (1988) A Calculus of Refinements for Program Derivation. *Acta Informatica*, **25**, 593–625.

[15] Dijkstra, E. W. and Scholten, C. S. (1989) *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin.

[16] Pan, S. (1995) *Software Quality Improvement, Specification Derivation and Quality Measurement Using Formal Methods*. Ph.D. Thesis, Griffith University, Australia.
[17] Pan, S. and Dromey, R. G. (1996) Reengineering Loops. *Comp. J.*, **39**, 184–202.
[18] Pan, S. and Dromey, R. G. (1993) *Loop Normalization*. Research Report, Griffith University, Australia.
[19] Alagic, S. and Arbib, M. (1978) *The Design of Well-structured and Correct Programs*. Springer-Verlag, New York.
[20] Farrow, P. (1993) Private communication. Centre of Information Technology Research, University of Queensland, Australia.