



Programming
Techniques and
Data Structures

Dr. Douglas Rector
Editor

A Generalized Control Structure and Its Formal Definition

DAVID LORGE PARNAS IBM Visiting Scientist, Federal Systems Division

David Parnas is interested in all aspects of software engineering including program semantics, process structure, and precise module specifications. He is currently leading an experimental redesign of a hard real time system in order to evaluate a number of software engineering principles. He is also involved in the design of a language involving new control structures and abstract data types.

Author's Present Address:
David Lorge Parnas,
Department of Computer
Science, University of
Victoria, P.O. Box 1700,
Victoria, British Columbia
V8W 2Y2, Canada. He is also
with the Computer Science
and Systems Branch of the
Naval Research Laboratory.
Permission to copy without
fee all or part of this material
is granted provided that the
copies are not made or
distributed for direct
commercial advantage, the
ACM copyright notice and
the title of the publication
and its date appear, and
notice is given that copying
is by permission of the
Association for Computing
Machinery. To copy
otherwise, or to republish,
requires a fee and/or specific
permission. © 1983 ACM
0001-0782/83/0800-0572 75¢

INTRODUCTION

This paper has three theses, each of which the author considers to be of importance to the professional programmer:

- (1) A precise and complete definition of a programming language control construct can be written using only the mathematics of sets and relations, mathematics that is commonly taught in high school.
- (2) A simple, general control structure can lead to programs that are at least as clear and efficient as programs that can be written with a battery of control structures, each designed to handle a restricted set of problems.
- (3) When developing programs it is often useful to develop and verify a nondeterministic algorithm and later to refine that algorithm to obtain an efficient deterministic program.

These theses are important to even the most pragmatic of programmers because, if true, they allow him or her:

- (a) to refuse to accept a reference manual that is "neither sufficiently precise nor sufficiently detailed" [1];
- (b) to reject languages that have such a confusing collection of "powerful control structures" that a programmer must begin by deciding which of the many features to use;
- (c) to develop algorithms that are relatively clear and easy to verify because they are not cluttered by unnecessary and arbitrary decisions that are forced on a programmer by a deterministic programming notation.

The paper consists of three sections:

- (1) Section I reviews the elementary algebraic concepts of sets and relations and then develops the mathematics that will be used to describe the behavior of programs and to define the meaning of control constructs;
- (2) Sections 2 and 3 explain how the mathematical structure can be interpreted as a model of program behavior and then use the mathematics to define a generalized version of Dijkstra's guarded commands [3].
- (3) Section 4 discusses programming using the new control structure. We include a short history of the development of control structures from a collection of complicated special cases to the simpler, more general structure presented here. With the aid of three examples, this section shows how the

ABSTRACT: A new programming language control structure as well as an improved approach to a formal definition of programming languages are presented. The control structure can replace both iteration and conditional structures. Because it is a semantic generalization of those structures, a single statement using the new control structure can implement the functions of loops, conditionals, and also programs that would require several conventional constructs. As a consequence of this increased capability, it is possible to write algorithms that are simpler, more efficient, and more clearly correct than those that can be written with earlier structured-programming control structures. In order to provide a precise definition of the new constructs, a new version of relational semantics, called LD-relations is presented. An algebra of these relations is developed and used to define the meaning of the new constructs. A short discussion of program development and the history of control structures is included.

generalized structure allows algorithms that are both more efficient and easier to understand than those that can be obtained using more restricted structures. Program development is also discussed in this section.

Many readers will be tempted to skip or skim the mathematical sections, believing them to be difficult and of little practical value. Although the fourth section can be read without reading the second section, it is not recommended. The second section appears more difficult than it actually is. If you understand set intersection, set union, and composition of functions, you can understand the algebra that is developed in Section 2. Without an understanding of the mathematics, you may not understand the control structure, and a complete understanding is required for the production of programs whose behavior we completely understand.

These ideas can be of use even to programmers who do not have the freedom to choose the language they will use. They can do their program development using the ideas in this paper (which are compatible with the methods presented by Dijkstra [3] and Gries [5]) and then translate these algorithms into the language available to them. Better programs will be their reward.

Appendix A presents a verification procedure for the new constructs. Appendix B contains the proofs of some simple theorems that were stated in Section 2.

AN ALGEBRA OF LIMITED DOMAIN RELATIONS

The following definitions are used in defining the control constructs. The first seven are standard and are included for completeness. Readers who prefer a more tutorial presentation should consult Stanat and McAllister [11].

Universe

In the following, we assume the existence of a set universe, U . All elements discussed below are members of U . We also assume that the concept of an ordered pair of elements from U is understood.

Relation

A relation R is a set of ordered pairs; both elements of the pair are members of U . We indicate that a pair (x, y) is in R by the notation, $(x, y) \in R$.

Domain

The domain of a relation R , denoted Dom_R , is the set of elements $x \in U$ such that R contains at least one pair with x as the first element. We write¹

$$x \in Dom_R \text{ iff } \exists y \in U: (x, y) \in R.$$

Total Relation

A relation is a total relation if its domain is U .

Limited Domain Relations (LD Relations)

A limited domain relation A is an ordered pair (R_A, C_A) where R_A is a relation and C_A is a subset of Dom_R . For any two LD relations² A and B ,

$$A = B \text{ iff } R_A = R_B \text{ and } C_A = C_B$$

$$A \subseteq B \text{ iff } R_A \subseteq R_B \text{ and } C_A \subseteq C_B.$$

An LD relation determines two relations of interest: R_A is the first component of A ; D_A is the set of ordered pairs (x, y) such that $x \in C_A$ and $y \in U$.

¹“:” should be read “such that.”

²“ \subseteq ” denotes set inclusion.

We call C_A the *competence set* of A . We define complement, intersection, and union for LD relations (denoted by $\sim, *, +$) using the corresponding operations on sets and relations. The set operations are denoted by \sim, \wedge, \vee , respectively. We also define composition for LD relations.

An Algebra of LD Relations

Complement. For any LD relation A such that $\sim(\exists x \in C_A \forall y[(x, y) \in R_A])$, the complement of A (denoted \bar{A}) is an LD relation such that $R_{\bar{A}}$ consists of all pairs in D_A that are not in R_A (i.e., $R_{\bar{A}} = D_A \wedge \sim R_A$) and $C_{\bar{A}} = C_A$.

Union. For any two LD relations, A and B , we define $A + B$ to be the LD relation such that an ordered pair is in R_{A+B} if it is in either R_A or R_B , i.e., $R_{A+B} = R_A \vee R_B$ and $C_{A+B} = C_A \vee C_B$.

Let A be a (possibly infinite) sequence of LD relations A_0, A_1, \dots . We define $(+_0)A$ as A_0 and $(+_i)A$ as $A_i + (+_{i-1})A$ for $i > 0$. We define $+A$ by

$$\begin{aligned} (x, y) \in R_{+A} & \text{ iff } \exists n : (x, y) \in R_{(+_n)A} \\ x \in C_{+A} & \text{ iff } \exists n : x \in C_{(+_n)A}. \end{aligned}$$

Intersection. For any two LD relations A and B such that $(Dom_{(R_A \wedge R_B)} \vee (Dom_{R_A} \wedge \sim C_B) \vee (Dom_{R_B} \wedge \sim C_A)) \supseteq C_A \vee C_B$, we define $A * B$ such that

- (1) a pair (x, y) is in R_{A*B} if one of the following holds: (a) it is in both R_A and R_B , or (b) it is in R_A and x is not in C_B , or (c) it is in R_B and x is not in C_A , i.e., $R_{A*B} = (R_A \wedge R_B) \vee (R_A \wedge \sim D_B) \vee (R_B \wedge \sim D_A)$.
- (2) $C_{A*B} = C_A \vee C_B$.

Let A be a (possibly infinite) sequence of LD relations A_0, A_1, \dots . We define $(*_0)A$ as A_0 and $(*_i)A$ as $A_i * (*_{i-1})A$ for $i > 0$. We define $*A$ by

$$\begin{aligned} x \in C_{*A} & \text{ iff } \exists n : x \in C_{(*_n)A} \\ (x, y) \in R_{*A} & \text{ iff } \exists n : \forall j \supseteq n : (x, y) \in R_{(*_j)A} \end{aligned}$$

Composition. Let A_1 and A_2 be LD relations. $A = A_1 \circ A_2$ is defined by

- (1) $(x, y) \in R_A$ iff $\exists z : [(x, z) \in R_{A_1} \wedge (z, y) \in R_{A_2}]$
- (2) $C_A = \{x : x \in C_{A_1} \wedge \forall y[(x, y) \in R_{A_1} \Rightarrow y \in C_{A_2}]\}$.

Readers should note that the convention for relational composition is not consistent with the convention used for functional composition. The order is reversed!

Some Theorems about the Algebra of LD Relations

Let A and B be LD relations such that $Dom_{R_A} = C_A$ and $Dom_{R_B} = C_B$; \bar{A} and \bar{B} are defined and $A * B$ is defined.

Theorem 0: $\bar{\bar{A}} = A$.

Theorem 1: $\bar{A * B} = \bar{A} + \bar{B}$.

Theorem 2: $A + B = \bar{(\bar{A} * \bar{B})}$.

Let A, B , and L be LD relations.

Theorem 3: $A + (B + L) = (A + B) + L$.

Theorem 4: $A + B = B + A$.

Theorem 5: $A * B = B * A$.

Theorem 6: $(A * B) * L = A * (B * L)$.

Theorem 7: $L \circ (A + B) \supseteq L \circ A + L \circ B$.

Theorem 8: $(L \circ A) \circ B = L \circ (A \circ B)$.

Proofs of these theorems are given in Appendix II.

PROGRAMS AND LD RELATIONS

We view a program as a nondeterministic mechanism that changes the state of the machine that it controls. We interpret the set U as the set of possible states of that machine. We denote the LD relation associated with a program P by L_P . If, when a program is started with the machine in state x , it is possible that it will terminate with the machine in state y , then (x, y) is a member of R_{L_P} . C_{L_P} is interpreted as the set of starting states in which P is guaranteed to terminate. We say "the domain of P ," meaning the domain of R_{L_P} .

The inclusion of (x, y) in R_{L_P} tells us only that y is a possible termination state when the program is started in state x ; it does not tell us that termination is guaranteed or that, if the program terminates, the termination state is certain to be y .

The set C_{L_P} represents the complete set of starting states in which termination of P is guaranteed. L_P provides a complete description of the semantics of P . Table I illustrates the meaning of the domain and competence sets.

In the sequel, unless otherwise stated, C may represent a proper subset of the states in which termination is guaranteed.

Limited Programs

We can write the program $g \rightarrow P$, where (1) g represents a predicate defined on U , (2) P represents a program, and (3) $g(x)$ implies that $x \in C_{L_P}$. $g \rightarrow P$ is called a limited program. Its associated LD relation $L_{g \rightarrow P}$ is the LD relation L_P restricted to those parts of the domain of R_{L_P} where g is true. The behavior of $g \rightarrow P$ when $\{x: g(x) \leq C_{L_P}\}$ is not true is undefined.

$$(x, y) \in R_{L_{g \rightarrow P}} \quad \text{iff} \quad g(x) \wedge (x, y) \in R_{L_P}$$

$$C_{L_{g \rightarrow P}} = \{x: g(x)\}.$$

It is the programmer's responsibility to guarantee that $\{x: g(x)\} \leq C_{L_P}$. Guards with this property are *proper guards*.

Sequential Execution

We can combine two programs P_1 and P_2 by writing $P_1; P_2$. This indicates a program in which execution of P_1 is followed by execution of P_2 . The semantics are given by

$$L_{P_1; P_2} = L_{P_1} \circ L_{P_2}.$$

TABLE I. The Meaning of Domain and Competence Sets

Behavior of Program P	Competence set C_{L_P}	Domain of R_{L_P}	R_{L_P}
P terminates when started in x	Includes x	Includes x	Includes (x, y) if P might terminate in y when started in x
P sometimes terminates when started in x	Does not include x	Includes x	Includes (x, y) if P might terminate in y when started in x
P never terminates when started in x	Does not include x	Does not include x	No pairs of the form (x, y)
P never terminates	Empty	Empty	Empty
P is never guaranteed to terminate but may	Empty	Nonempty	Includes (x, y) if P might terminate in state y when started in state x

Limited-Program Lists

The limited-program lists defined below provide two additional ways to combine limited programs. The program segment

$$g_1 \rightarrow p_1 \vee g_2 \rightarrow p_2 \vee \dots \vee g_n \rightarrow p_n \quad (P_s)$$

is called a *select list*. It represents a program P_s , whose associated LD relation is the union of the LD relations of the components $L_{P_s} = +L$, where $L_i = L_{g_i \rightarrow p_i}$.

The program segment

$$g_1 \rightarrow p_1 \& g_2 \rightarrow p_2 \& \dots \& g_n \rightarrow p_n \quad (P_j)$$

is called a *jury list*. It represents a program P_j , whose associated LD relation is the intersection of the LD relations of the components $L_{P_j} = *L$, where $L_i = L_{g_i \rightarrow p_i}$.

The Iterative Control Structure

The iterative control structure consists of the brackets **it** and **ti** surrounding a limited-program list in which each limited program is followed by either \uparrow or \downarrow . We define the semantics of **it ti** by showing how to determine the LD relation of the construct from the LD relation of the components.

Let L^* be the LD relation of the limited-program list comprising the components that are followed by a \downarrow . Let L' be the LD relation of the limited-program list comprising the components that are followed by \uparrow . Let L_0 be defined by

$$R_{L_0} = R_{L^*}.$$

$$C_{L_0} = C_{L^*} \wedge \sim C_{L'}.$$

Let H be the sequence L_0, L_1, L_2, \dots , where $L_i = L_0 + L' \circ (L_{i-1})$ for $i > 0$. C_{L_i} is the set of states in which one of the programs marked with a \downarrow is certain to be selected. R_{L_0} describes the possible starting and stopping states for a limited-program list that consists only of the \downarrow components.

C_{L_i} is the set of states in which there can be at most i executions of programs marked by a \uparrow and then a program marked by a \downarrow will be selected. R_{L_i} describes the possible starting and stopping states when there are at most i executions of \uparrow programs followed by an execution of a \downarrow program.

The LD relation of the **it ti** construct is given by

$$L_{\text{it ti}} = +H. \quad (1)$$

Using Theorem 7 to show that $L_{i+1} \supseteq L_i$, we can write

$$L_{\text{it ti}} = L_0 + L' \circ (L_0 + L' \circ (L_0 + L' \circ (\dots))) \quad (2)$$

or

$$L_{\text{it ti}} = L_0 + L' \circ L_{\text{it ti}}. \quad (3)$$

Eq. (3) is a recursive description of the semantics of the **it ti** statement. It describes a mechanism that either "executes" L_0 and terminates or "executes" L' and then "executes" the **it ti** statement. Verification that an LD relation proposed for an **it ti** statement is correct is discussed in Appendix A.

An Interpretation for Programmers

The algebra of LD relations is intended as a means of describing the behavior of nondeterministic programs. It may also be used to write specifications for deterministic programs where those specifications allow several externally distinguishable deterministic implementations.

In general, a nondeterministic program P , started in state x , may terminate in one of several states $(y_1, y_2, y_3, \dots, y_n)$ or not terminate. R_{L_P} must contain the pair (x, y_i) for each such y_i . If $x \in C_L$ when P is started in state x , termination is guaranteed.

If a program is started in a state that is not in the program's domain, the program behaves as if it had entered a nonterminating loop. There is no terminating state. Following Dijkstra [3], we term this *abortion*. Abortion in this sense is not abnormal termination. Only by examination of the implementation could we learn what the program "actually does" when abortion occurs.

When a program built using the control constructs described above is started in a state that is in its competence set, each of the component programs will only be started in states within their competence sets. This is sufficient to prevent abortion or nontermination. Recall, however, that it is the programmer's responsibility to include proper guards.

The complement of L_P describes another program P' such that if P' is started in state x , it may terminate in any state in which P cannot terminate if P is started in state x . However, because the domain of P' will not include any state x such that $\forall y[(x, y) \in R_{L_P}]$, the complement of such programs does not exist.

A guard limits the use of the program to a (possibly) smaller domain. A limited program will abort if it is started in a state where the guard is false or in any state outside the program's domain.

The symbol ";" denotes the sequential execution of two programs. Sequential execution implements relational composition in those states where the state resulting from executing the first program is guaranteed to be in the competence set of the second program. The definition of composition for LD relations restricts the competence set of the composite relation to such states.

The limited-program lists provide two additional ways of combining programs:

A *select list* indicates that one of the components of the list is to be chosen. For any given starting state, a component is eligible for selection only if the state is in its competence set, i.e., if its guard is true in that state. If two or more components are eligible for selection, an arbitrary choice is made. If the select-list program is started in a state that is outside the competence set of all of its components, the execution will abort.

A *jury list* demands agreement among the components that are selected. Each of those programs may allow one or more end states. If they do not all agree on at least one, the execution will abort. If they agree on several end states, any of those states may be selected.

The *it ti* replaces both conventional loops and conditional statements. It repeatedly invokes programs whose guard evaluates to **true** until one of those marked with a \downarrow is selected. After that program is executed, iteration is terminated. If both terminating (\downarrow) and nonterminating (\uparrow) components are eligible, the choice is not specified. In the case of the jury list, the eligible component programs that call for termination must agree with each other on an end state; those that call for continuation must also agree with each other. However, those that call for termination need not agree with those that do not. If the *it ti* is executed in a state where no guard is true, abortion occurs.

On Termination

The following paragraphs illustrate the way the mathematics allows the precise description of even the more subtle aspects of the behavior of nondeterministic programs.

The semantic definition of *it ti* allows free choice if both a \uparrow component and a \downarrow component are eligible for selection. Recall that C_L indicates those states in which termination after at

most i executions of a \uparrow component followed by execution of a \downarrow component is guaranteed. It follows that C_L is the set of states in which execution of a \downarrow component followed by immediate termination is guaranteed. In the definition of *it ti*, C_L does not include any states in which a \uparrow component is eligible for selection. Thus, when both are eligible, the implementation may make an arbitrary choice between \uparrow and \downarrow components.

In an earlier version [10], we defined $L_0 = L^*$. C_L included all of the states in which a \downarrow component was eligible, whether or not a \uparrow component was eligible. Under that definition, the iteration would stop as soon as possible, and the competence set of that statement might include states in which termination is not guaranteed by the present definition.

There is a pragmatic justification for either definition! The original definition would terminate as soon as possible with obvious practical advantages. However, the implementation would have to evaluate all \downarrow guards before selecting a \uparrow alternative. This would be a disadvantage that might outweigh the obvious advantage. The reader should note that there are programs whose termination is guaranteed under the alternative interpretation but not under that chosen in this paper. This paper's definition allows the implementor to be consistent with the other definition.

It is also possible to specify that the program continue as long as possible. This is left as an exercise for readers who are in the employ of manufacturers.

The reader should note that the LD relation of a program deals with both "partial correctness" and termination. Once one has determined the LD relation of a program, no additional study of termination is needed.

The Programmer's Responsibility to Limit Executions of Programs

The implementation of a programming notation cannot be expected to determine the domain of a program or when execution of the program will lead to abortion or nontermination. It is the responsibility of the programmer to write programs in such a way that programs are not executed in states where such things can happen. The guards are his tools for that task.

Every programmer begins with a set of previously written programs and uses them to construct new programs. He should be able to assume that the information supplied with each program properly describes the LD relation of that program. He must then determine guards for the programs that he writes. A proper guard adheres to the following rule:

For the limited program $g \rightarrow P$: $g(x) \Rightarrow x \in C_{L_P}$.

Although it is possible to define the semantics of programs in which this rule is violated, this paper has not done so. The semantics given in this paper assume proper guards.

Why Program Using *it ti*?

Several goals have motivated the design of the new programming constructs.

Improved Programs. There are programming problems where the use of the so-called structured concepts require the introduction of extra variables and/or additional evaluations of expressions. In such cases, the extra variables have been found to be a source of errors and the purpose of a program is often obscured [4]. The *it ti* construct allows the elimination of some of those variables and leads to simpler programs. Illustrations are provided below.

Stricter Programming Discipline. In the use of Dijkstra's *if fi* [3] we see the imposition of a useful discipline. It is necessary to make sure that one has specified an action for all states in which the program might be started. An *if fi* program aborts if it is started in a state where none of the guards is true. In Dijkstra's *do od* this discipline is not applicable. If a state has been forgotten, the loop will simply terminate with incorrect results. The *it ti* construct imposes the *if fi* discipline uniformly. Termination conditions (and actions upon termination) must be specified just as the continuation conditions (and associated actions) are specified.

Generalized Constructs. There is so much similarity between the formal definitions of conditionals and loops, that it is appealing to have a single construct that fulfills both needs. The conditional (*if fi*) is a special case of *it ti* in which all of the elements of the limited-program list have a \downarrow affixed. The loops differ from the *do od* only in requiring an explicit termination condition. However, the generalization allows one-statement *it ti* programs that do the work of two or more of the more restricted constructs. This is illustrated below.

Mills' *while* existence theorem [9] describes fundamental limitations on the class of programs that can be implemented as a single loop of either the *while do* form or the *do od*. Because of the way *it ti* generalizes conditionals and loops, this limitation does not apply.

A Structure for Reliable, Redundant Programs: The Jury List. Some artificial intelligence programs are written in a way that makes the jury list a natural structure. There are also situations where requiring agreement of separately written programs is useful for increased reliability through redundant computations. In those cases, one may use the jury construct directly. However, Theorems (0)–(2) show that, in theory, we do not need both constructs. These theorems allow us to replace any program with a semantically equivalent program that uses only one of the constructs. Although the duality is appealing, the semantically equivalent programs are not necessarily equivalent when efficiency and reliability are considered. The use of the jury list and Theorems (0)–(2) are subjects for further study. The jury list is not discussed further in this paper.

More Efficient Implementation. A preliminary study suggests that programs using these constructs can be implemented more efficiently than is practical if the program is written in terms of the extra variables and expression evaluations needed using older constructs.

These points are illustrated in a later section, which gives examples of the use of the new constructs and compares them with older constructs.

DETERMINISTIC PROGRAMS

If we do not wish to allow nondeterminism in the constructs, we can make the following changes:

- (1) All relations are functions. The competence set and domain are identical.
- (2) We can allow \vee to be replaced by *else* with the semantics that each guard be implicitly conjoined with the negation of the union of the previous guards in the list. In other words, the statement

$$\text{it } A \rightarrow P_1 \text{ else } B \rightarrow P_2 \text{ else } C \rightarrow P_3 \dots \text{ ti}$$

is equivalent to

$$\text{it } A \rightarrow P_1 \vee (\text{not } A \text{ and } B) \rightarrow P_2 \vee ((\text{not } (A \text{ or } B))) \text{ and } C \rightarrow P_3 \dots \text{ ti.}$$

(3) In this context, the jury list and the select list have the same meaning because the competence sets of the programs are disjoint.

It is often possible to make a program appear simpler by taking advantage of deterministic sequencing to eliminate duplicate predicates from the guards in an *it ti*. However, such programs are often harder to understand and verify for two reasons:

- (1) The correctness of the individual components of an *it ti* will depend on the guards that precede them; in the nondeterministic form the correctness is independent and each limited-program can be understood and verified separately.
- (2) In the deterministic form we are often forced to make arbitrary decisions, thereby destroying a symmetry or duality that might make the program easier to understand.

The nondeterministic forms are convenient for deriving programs following Dijkstra's method of writing guards that are the weakest preconditions for the execution of the program with the desired end result [3, 5]. This process often results in guards that are not mutually exclusive. The resulting nondeterministic program includes only correct computations, but may be less efficient than a deterministic program would be because one cannot exploit knowledge about the likelihood of certain predicates being true when the statements are executed. The step from nondeterministic program to deterministic program may be viewed as an efficiency improvement step made with the confidence that one is selecting a computation from a set of correct computations.

Exploiting Information about Initial Conditions: *init*

Programmers who require the utmost efficiency are often not satisfied with any loop construct that requires a test before execution of the statements in the body. Often the programmer has knowledge about the initial conditions of the loop that would allow him to skip an initial test or to skip an action on the first iteration of a loop. If the action to be performed initially is the same as an action that is conditionally performed within the iteration statement, iteration structures in which there is always an initial test force the programmer to choose between either performing the unnecessary test on the first iteration or duplicating the code outside the loop.

The source of this problem is not that the control structure requires the programmer to write a predicate describing the conditions under which each action should be performed. Rather, it is a weakness of the set of predicates available to the programmer for expressing those conditions. Standard programming languages provide no way to express the predicate "initial iteration" except by introducing a variable that is set before the loop begins and reset after execution of each nonterminating component.

We can correct this deficiency by adding a predicate *init* to the primitive predicates that can be used for writing guards in the *it ti* statement. Informally, the semantics are that *init* is *true* initially for the loop, i.e., when control passes to the *it ti* from the previous statement, but will be evaluated as *false* after the first execution of any component of the *it ti*. *init* can be used either by itself or as part of a guard expression.³

³ If *init* is used in the form *init* *cor* *boolean expression*, the understanding is that Boolean expression will not be evaluated the first time through the loop. If the left-hand operand of *cor* is true, the value of the right-hand operand need not be defined.

```

proc makedeed (alt output: fix input)
  var input: sequence of char  {input characters}
  var output: string           {output string}
  var data: char               {temporary—for reading
                               character}
  var d, e: integer            {temporary—for counting
                               D's, E's}
  var found: boolean           {temporary—for indicating
                               success}

  do
    d, e, found := 0, 0, false
    reset input
    while
      input ≠ empty ∧ found = false
    do
      data := next(input)
      case
        data
      part ('D')
        d := d + 1
      part ('E')
        e := e + 1
      esac
      if
        d ≥ 2 ∧ e ≥ 2
      then
        found := true
      fi
    od
  od
  if
    found = true
  then
    output := output ↑↑ 'DEED'
  else
    output := output ↑↑ 'NO DEED'
  fi
corp

```

FIGURE 1. Solution Using PDL [6]. ("↑↑" denotes concatenation; the programming language is defined in [6].)

The effect of the introduction of **init** into guards within an **it ti** is defined in terms of LD relations. Let L_0 be the relation of the list of terminating components computed assuming that **init** is **false**, LL_0 be the relation of the list of terminating components computed assuming that **init** is **true**, L' be the relation of the nonterminating components of the list assuming that **init** is **false**, LL' be the relation of the nonterminating components of the list assuming that **init** is **true**, and

$$L_{it\ ti} = L_0 + L' \circ (L_0 + L' \circ (L_0 + L' \circ (\dots))). \quad (1)$$

The LD relation of **it ti** statements in which **init** appears is given by

$$LL = LL_0 + LL' \circ L_{it\ ti}. \quad (2)$$

If there is no occurrence of **init**, LL reduces to $L_{it\ ti}$ by using

$$LL_0 = L_0, \quad LL' = L', \quad \text{and} \quad L_0 + L' \circ L_{it\ ti} = L_{it\ ti}.$$

The implementation of statements using **init** does not usually require the introduction of a variable to store the value of **init**. If the use of **init** is confined to guards of the form **init or predicate**, **init cor predicate**, **(not init) and predicate**, and **(not init) cand predicate**, the object code can jump directly to the statements that should be executed, then return to the start of the loop code, which is compiled as if **init** were **false**.

Illustrations of the Use of the **it ti** Construct

First Example. The first example is taken from Forthofer [4]. The problem is to search an unordered input sequence of characters looking for the characters that constitute "DEED," i.e., to search for two occurrences of "D" and two occurrences of "E" in any order. If the search is successful, the output should be "DEED"; otherwise, it should be "NO DEED." Figure 1 is the solution in PDL ([6]) that is given in [4]. Figure 2 gives the solution using the nondeterministic constructs introduced in this paper. Components of the guards are repeated because of the nondeterministic constructs. Figure 3 gives a version of this program in which we have used the deterministic constructs. One can adopt the convention of leaving out true guards. We have not.

In both its deterministic and nondeterministic versions, this example illustrates the basic advantage of the new constructs. In the PDL version, the variable *found* was used to transfer information from the loop to a final conditional statement that determined the output. By combining the two statements, we eliminate the variable and simplify the program while allowing a more efficient, compiled translation.

One can get the same algorithm with Dijkstra's guarded commands, but it requires simulating the terminating constructs by introducing a Boolean variable *done* into the program, including it in the guards of the loop, setting it to **true** when ready to terminate, and —we almost forgot—initializing it to **false**. Figure 4 shows that version. It is obviously not a program that anyone would write except to illustrate the method used in Figure 2.

```

var input: sequence of char
var output: string
var data: char
var d, e: integer
d, e := 0, 0;
reset input;

it (input ≠ empty) ∧ ((d < 2) ∨ (e < 2)) → data := next (input);
  it data = 'D' → d := d + 1 ↓
    ∨ data = 'E' → e := e + 1 ↓
    ∨ data ≠ 'E' ∧ data ≠ 'D' → skip ↓
  ti ↑
  ∨ (d ≥ 2) ∧ (e ≥ 2) → output := output ↑↑ 'DEED' ↓
  ∨ (input = empty) ∧ ((d < 2) ∨ (e < 2)) → output := output ↑↑ 'NO DEED' ↓
ti

```

FIGURE 2. Solution of DEED Using **it ti**.

```

var input: sequence of char
var output: string
var data: char
var d, e: integer
d, e := 0, 0;
reset input;

it (d ≥ 2 ∧ e ≥ 2) → output := output ↑↑ 'DEED' ↓
else (input = empty) → output := output ↑↑ 'NO DEED' ↓
else true → data := next (input);
  it data = 'D' → d := d + 1 ↓
  else data = 'E' → e := e + 1 ↓
  else true → skip ↓
ti ↑
ti

```

FIGURE 3. Deterministic Solution Using **it ti**.

```

var input: sequence of char
var output: string
var data: char
var d, e: integer
var done: boolean
d, e := 0, 0;
done := false;
reset input;

do ~done ∧ (input ≠ empty) ∧ ((d < 2) ∨ (e < 2)) → data := next
(input);
if data = 'D' → d := d + 1
□ data = 'E' → e := e + 1
□ data ≠ 'E' ∧ data ≠ 'D' → skip
fi
□ ~done ∧ (d ≥ 2) ∧ (e ≥ 2) → output := output ↑↑ 'DEED';
done := true
□ ~done ∧ (input = empty) ∧ ((d < 2) ∨ (e < 2)) → output := output
↑↑ 'NO DEED'; done := true
od

```

FIGURE 4. Algorithm of Figure 3 Translated to do od and if fi.

Second Example. As a second example, we consider the pattern-matching program developed by Dijkstra ([3], ch. 18). Figure 5 shows Dijkstra's code for the main portion of the program. We do not show the code that initializes d as it is not relevant here. The program searches for a pattern represented by P in a string represented by x . Whenever the search stops, the position in the string r and the position in the pattern k are adjusted by $d(k)$.

A direct application of the new programming constructs is to combine the inner loop with the subsequent if statement, as shown in Figure 6. This simplifies the control structure of the program a little, but at the cost of repeating the pattern-matching predicate because of the nondeterminism. An "optimizing" compiler could compile efficient code from this version by noting the repeated predicates.

```

begin glocon p, N, x, M; virvar count; privar r, k; pricon d;
"initialize d";
count var int, r var int, k var int := 0, 0, 0;
do r ≤ M - N →
  do k ≠ N and p(k) = x(r + k) → k := k + 1 od;
  if k = N → count := count + 1; r, k := r + d(k), k - d(k) □
    0 < k < N → r, k := r + d(k), k - d(k) □
    k = 0 → r := r + 1
  fi
od
end

```

FIGURE 5. Dijkstra's Pattern Match Program.

```

it r ≤ M - N →
  it k ≠ N and P(k) = x(r + k) → k := k + 1 ↑
  ∨ k = N → count := count + 1; r, k := r + d(k), k - d(k) ↓
  ∨ k = 0 and P(k) ≠ x(r + k) → r := r + 1 ↓
  ∨ 0 < k < N and P(k) ≠ x(r + k) → r, k := r + d(k), k - d(k) ↓
  ti ↑
  ∨ r > M - N → skip ↓
ti

```

FIGURE 6. if ti Program Combining Loop and Conditional.

```

it r > M - N → skip ↓
∨ (r ≤ M - N) ∧ (k ≠ N) and P(k) = x(r + k) → k := k + 1 ↑
∨ (r ≤ M - N) ∧ (k = N) → count := count + 1; r, k := r + d(k),
  k - d(k) ↓
∨ (r ≤ M - N) ∧ (k = 0) and P(k) ≠ x(r + k) → r := r + 1 ↑
∨ (r ≤ M - N) ∧ (0 < k < N) and P(k) ≠ x(r + k) → r, k := r + d(k),
  k - d(k) ↓
ti

```

FIGURE 7. Revision of Figure 6 Combining Nested Loops.

```

it r ≤ M - N →
  it k = N → count := count + 1; r, k := r + d(k), k - d(k) ↓
  else P(k) = x(r + k) → k := k + 1 ↑
  else k = 0 → r := r + 1 ↓
  else true → r, k := r + d(k), k - d(k) ↓
  ti ↑
  else true → skip ↓
ti

```

FIGURE 8. Deterministic Revision of Figure 6 (Nested Loops).

```

it r > M - N → skip ↓
else k = N → count := count + 1; r, k := r + d(k), k - d(k) ↑
else P(k) = x(r + k) → k := k + 1 ↑
else k = 0 → r := r + 1 ↑
else true → r, k := r + d(k), k - d(k) ↑
ti

```

FIGURE 9. Deterministic Revision of Figure 7 (Single Loop).

The simplification reveals that we can combine the two loops, as shown in Figure 7. Here the structure is much simpler; the correctness could be more quickly proven, but the problem of duplication of predicates has gotten worse.

However, this is a problem in which nondeterministic choice of component programs does not arise. Figures 8 and 9 show versions of the program using the **elseor** notation. These simple programs include each predicate only once. I do not believe that conventional structured programming control structures or Dijkstra's control structures allow a program of this sort. The simplification is only possible with control structures that combine iteration and conditional execution. Figure 8 uses two nested loops, thereby testing r as infrequently as possible. The version in Figure 9 is a bit simpler and, if the likelihood of a match is low, it would be almost as efficient. In some implementations it would require less space for code than the program in Figure 8.

Third Example. Figure 10 shows a small section of code from a real-time system. The instruction **READ** fails occasion-

```

ct := 0;
success := false;
while not success and ct lt numtry do
  READ(/AOA/, raw_aoa, success);
  ct := ct + 1;
endwhile;
if success then
  true_aoa := 0.76 * (greal(raw_aoa, -3) / scale_offset) - 8.68 fi

```

FIGURE 10. Real-Time Input: Conventional Program.

ally. The Boolean variable *success* is made **false** when it fails; otherwise *success* will be set to **true**. The program is required to attempt *READ* at most *numtry* times. If there is no success after that, the program will use an old value and signal its failure to other programs. This loop is an inner loop in a process that runs periodically. The variables *ct* and *success* are initialized to 0 and **false**, respectively. The program can reset *ct* either before or after using it, but the value of *success* is used in subsequent code.

Defenders of the use of **go to** would point out two problems in the algorithm in Figure 10. The program tests *success* twice (once to terminate the loop and again for the conditional statement) when only one test is necessary. Further, the initialization of *ct* and then the initial test of *ct* and *success* are not necessary. The initial *READ* can be done without any test.

Figures 11(a) and 11(b) show two alternative versions of the program using the **it ti**. The unnecessary test of success has been eliminated. However, in Figure 11(a) the *READ* command appears twice, and in Figure 11(b) *success* is set to **false** and then tested unnecessarily in order to eliminate that duplication.

Figures 12(a) and 12(b) show two alternative solutions using the **init** predicate in the guards. Figure 12(a) is the most straightforward use of **init**. The only problem is that, with a simple implementation of the constructs, *success* would be tested twice whenever it was **true**. It should be noted that by use of **init cor** we have eliminated the need to reset *success* to **false**. It will not be tested on the initial iteration.

In Figure 12(b) we have reversed the order of the two components in order to eliminate the second occurrence of success. We have also used (**not init**) **cand success** to eliminate the initial test of *success*. An implementation of this program can jump directly to the middle of the loop just as if the program had been written with a **go to**.

```

—read until success but at most numtry times
ct:=1;
READ(/AOA/, raw-aoa, success)
it success →
  true-aoa := 0.76 * (greal(raw-aoa, -3)/scale-offset) - 8.68 ↓
else ct lt numtry →
  READ(/AOA/, raw-aoa, success);
  ct := ct + 1 ↑
else true → skip ↓
ti

```

FIGURE 11(a). *it ti* Program with Duplicate Code.
success := **false**;

```

—read until success but at most numtry times
it success →
  true-aoa := 0.76 * (greal(raw-aoa, -3)/scale-offset) - 8.68 ↓
else ct lt numtry →
  READ(/AOA/, raw-aoa, success);
  ct := ct + 1 ↑
else true → skip ↓
ti;
ct:=0

```

FIGURE 11(b). *it ti* Program with Redundant Test.

```

it init cor (not success and ct lt numtry) →
  READ(/AOA/, raw-aoa, success);
  ct := ct + 1 ↑
else success →
  true-aoa := 0.76 * (greal(raw-aoa, -3)/scale-offset) - 8.68 ↓
else true → skip ↓
ti;
ct:=0

```

FIGURE 12(a). *it ti* Program Using **init**

```

it (not init cand success) →
  true-aoa := 0.76 * (greal(raw-aoa, -3)/scale-offset) - 8.68 ↓
else init or ct lt numtry →
  READ(/AOA/, raw-aoa, success);
  ct := ct + 1 ↑
else true → skip ↓
ti;
ct:=0

```

FIGURE 12(b). *it ti* Program Using **init** and **not init**

AN ABBREVIATED HISTORY OF CONTROL STRUCTURES

Early programmers used one of several forms of the **go to** to provide connections between program segments. Although some forms of the **go to** do impose restrictions on the structure of programs, few program writers complained about those restrictions. The complaints came from program readers. They were unable to recognize commonly occurring structures without careful study of the program. In response to such complaints, languages began to include control structures that provided those commonly occurring cases. Examples include the three-way branch of Fortran (motivated in part by the three-way branch found in some hardware) and the **for** statement in Algol 60. Because these special case structures imposed restrictions that some found unacceptable, the **go to** was retained. Because the use of the **go to** can even make the structure of well-structured programs hard to discover, there has been growing pressure to avoid the use of the **go to**. It has been found that more general control structures such as the **if then else**, **do while**, and **do until** allow easily recognized programs but were less restrictive than the structures that were designed to handle the most commonly occurring special cases. However, as Maddux [7] has shown us, there remains an infinite set of prime programs—programs that cannot be achieved with those constructs.

A significant step in the direction of more general structures was taken by Dijkstra [3]. His **do od** and **if fi** were each considerably less restrictive than the well-known earlier structures. Unlike **go to** programs, his programs are easily understood because the syntax makes the structure explicit. However, because each structure is restricted to either iteration or conditional execution, some restrictions remain. The structures developed in this paper are a further step toward generalization; by combining iteration and alternation we have removed restrictions. The syntax continues to make the structure apparent. The programs are still easily understood.

The **it ti** as presented here is still restricted. Some suggestions about the nature of the restrictions are to be found in Mills [8]. Further generalizations remain a subject for future study.

CONCLUSIONS

Workers in the field of programming methodology seek notations and disciplines that allow them to write programs whose structure is so clear that their correctness (or lack of it) becomes obvious. It has been found that whenever the understanding of a program requires explicit discussion of the execution sequence of statements in that program, one's ability to properly comprehend the program is reduced. That is why most workers in the field eschew explicit transfer of control in the form of the **go to**. The use of "structured" control constructs allows us to systematically analyze a program without explicit consideration of the execution sequence. Unfortunately, the use of these sometimes requires the introduction of extra variables and assignments which, in turn, make it more difficult to understand the program and increase the likelihood of error.

This paper has introduced and discussed a single control construct, the **it ti**. A single **it ti** can implement functions that would otherwise require a combination of loops and conditional statements. We can eliminate the variables that would be used to transmit information between program segments. With a few examples, the paper has demonstrated how the more general construct can increase both the efficiency and the clarity of the programs. We hope this reduces the number of people who resist structured programming ideas because of their fear of inefficiency. The paper also shows that this improvement in efficiency and clarity can be achieved without complicating either the semantic definition or the verification of the programs.

This paper has also introduced a predicate, **init**, that can be used to improve efficiency and clarity of iteration programs that exploit a programmer's knowledge of the initial conditions. There is a simple formal definition of the predicate.

In working with these constructs we sometimes discovered we could eliminate sequencing altogether, removing the ":", by introducing extra variables in an **it ti** statement. This brings us closer in form to the oft-touted ideas of functional programming languages. However, those programs are less efficient and their structure is less clear than that of the programs using the ":". I suspect that as long as we are dealing with sequential machines, ":" will have a place in our programming languages.

Since discovering these constructs, we have rewritten a number of well-designed algorithms that were originally written using older constructs. We have consistently been able to improve both the clarity and efficiency of those programs and hope that our readers find the ideas equally useful.

APPENDIX A: VERIFICATION OF **IT TI** STATEMENTS

The formula for the semantics of the statement does not always provide a closed-form expression for L . However, in a top-down programming methodology one would not write a statement and then try to find the corresponding LD relation. One should have the LD relation and then look for a program. The definition provides us with a way to verify that the statement has the desired relation.

Our definition of the LD relation for the **it ti** statement is

$$L = L_0 + L' \circ (L_0 + L' \circ (L_0 + L' \circ (\dots)))$$

where L_0 and L' have been defined earlier in the paper.

We see that

$$L = L_0 + L' \circ L. \quad (A1)$$

Since L_0 and L' can be determined from the program text, Eq. (A1) can be used as a check on L . It is clear from the above that Eq. (A1) is a necessary condition on L . We must

now ask under what conditions Eq. (A1) is both necessary and sufficient. We have not shown that Eq. (A1) has a unique solution.

Let us assume that we have two distinct solutions to Eq. (A1), L_1 and L_2 . $L_1 - L_2$ is the symmetric difference of L_1 and L_2 , i.e., $(R_{L_1} - R_{L_2}, C_1 - C_2)$. It describes behavior included in one solution, but not in the other. Using Eq. (A1) we see that

$$L_1 - L_2 = L' \circ L_1 - L' \circ L_2. \quad (A2)$$

$$\text{Since } L \circ A - L \circ B \subseteq L \circ (A - B), \quad (A3)$$

$$L_1 - L_2 \subseteq L' \circ (L_1 - L_2). \quad (A4)$$

Eq. (A4) tells us that if $R_{L_1 - L_2}$ contains a pair (x, y) , then there exists an infinite sequence z_1, z_2, \dots , such that

$$\begin{array}{ll} (x, z_1) \in R_{L'} & (z_1, y) \in R_{L_1} - R_{L_2} \\ (z_1, z_2) \in R_{L'} & (z_2, y) \in R_{L_1} - R_{L_2} \\ \vdots & \vdots \end{array}$$

From this we see that termination of the loop is not guaranteed if it is started in state x . Thus, for any loop where we can prove termination, Eq. (A1) has a unique solution; if we find an LD relation that satisfies Eq. (A1) we have the LD relation of the statement. However, with this method, the proof of termination must be done separately.

APPENDIX B: PROOFS OF THEOREMS 0-6

In the following we use the definitions of the operations on LD relations to prove the analogs of standard theorems. We carry the set C along, explicitly using a comma to separate the relation of an LD relation from the competence set.

Theorem 0

If $\text{Dom}_{R_A} = C_{R_A}$ and \bar{A} is defined,

$$\begin{aligned} A &\stackrel{?}{=} \bar{A} \\ &\stackrel{?}{=} \overline{D_A \wedge \sim R_A, C_A} \\ &\stackrel{?}{=} D_A \wedge \sim (D_A \wedge \sim R_A), C_A \\ &\stackrel{?}{=} D_A \wedge (\sim D_A \vee R_A), C_A \\ &\stackrel{?}{=} D_A \wedge R_A, C_A \\ &\stackrel{?}{=} R_A, C_A \\ &= A. \end{aligned}$$

Theorem 1

If $\text{Dom}_{R_A} = C_{R_A}$ and $\text{Dom}_{R_B} = C_{R_B}$, \bar{A} is defined, \bar{B} is defined, and $A * \bar{B}$ is defined,

$$\begin{aligned} A * \bar{B} &\stackrel{?}{=} \overline{(\bar{A} + \bar{B})} \\ &\stackrel{?}{=} \overline{(D_A \wedge \sim R_A) \vee (D_B \wedge \sim R_B), C_A \vee C_B} \\ &\stackrel{?}{=} (D_A \vee D_B) \wedge ((\sim D_A \vee R_A) \wedge (\sim D_B \vee R_B)), C_A \vee C_B \\ &\stackrel{?}{=} (D_A \vee D_B) \\ &\quad \wedge (\sim D_A \wedge \sim D_B \vee \sim D_A \\ &\quad \wedge R_B \vee \sim D_B R_A \vee R_B \wedge R_A), C_A \vee C_B \\ &\stackrel{?}{=} R_A \wedge \sim D_B \vee R_B \wedge R_A \vee \sim D_A \wedge R_B, C_A \vee C_B \\ &= A * B. \end{aligned}$$

Theorem 2

If $\text{Dom}_{R_A} = C_{R_A}$ and $\text{Dom}_{R_B} = C_{R_B}$, \bar{A} is defined, \bar{B} is defined, and $A \star B$ is defined,

$$\begin{aligned} \overline{(A \star B)} &\stackrel{?}{=} A + B \\ \overline{(\bar{A} + \bar{B})} &\stackrel{?}{=} \quad \text{by Theorem 1} \\ \bar{A} + \bar{B} &\stackrel{?}{=} \quad \text{by Theorem 0} \\ A + B. \end{aligned}$$

Theorem 3

$$\begin{aligned} A + (B + L) &= (A + B) + L \\ A + ((R_B \vee R_L), C_B \vee C_L) &= \\ R_A \vee R_B \vee R_L, C_A \vee C_B \vee C_L &= \\ (R_A \vee R_B) \vee R_L, (C_A \vee C_B) \vee C_L &= \\ (A + B) + L &= \end{aligned}$$

Theorem 4

$$A + B = B + A$$

which follows directly from symmetry of definition.

Theorem 5

$$A \star B = B \star A$$

which follows directly from symmetry of definition.

Theorem 6

$$(A \star B) \star L = A \star (B \star L).$$

$$\begin{aligned} \text{(a)} \quad C_{(A \star B) \star L} &= (C_A \vee C_B) \vee C_L = C_A \vee (C_B \vee C_L) = C_{A \star (B \star L)} \\ \text{(b)} \quad R_{(A \star B) \star L} &= (R_A \wedge R_B \vee R_A \wedge \sim D_B \vee R_B \wedge \sim D_A) \wedge R_L \\ &\quad \vee (R_A \wedge R_B \vee R_A \wedge \sim D_B \vee R_B \wedge \sim D_A) \wedge \sim D_L \\ &\quad \vee R_L \wedge \sim (D_A \vee D_B) \\ &= R_A \wedge R_B \wedge R_L \vee R_A \wedge R_L \wedge \sim D_B \\ &\quad \vee R_B \wedge R_L \wedge \sim D_A \vee R_A \wedge R_B \wedge \sim D_L \\ &\quad \vee R_A \wedge \sim D_B \wedge \sim D_L \vee R_B \wedge \sim D_A \wedge \sim D_L \\ &\quad \vee R_L \wedge \sim D_A \wedge \sim D_B \\ &= R_A \wedge (R_B \wedge R_L \vee R_L \wedge \sim D_B \vee R_B \wedge \sim D_L) \\ &\quad \vee (R_B \wedge R_L \vee R_B \wedge \sim D_L \vee R_L \wedge \sim D_B) \wedge \sim D_A \\ &\quad \vee (R_A \wedge \sim D_B \vee D_L) \\ &= R_{A \star (B \star L)}. \end{aligned}$$

Theorem 7 ("." denotes standard relational composition.)

$$\begin{aligned} L \circ (A + B) &\supseteq L \circ A + L \circ B. \\ \text{(a)} \quad C_{L \circ (A+B)} &= \{x: x \in C_L \wedge \forall y[(x, y) \in R_L \Rightarrow y \in C_{(A+B)}]\} \\ C_{L \circ A + L \circ B} &= \{x: x \in C_L \wedge (\forall y[(x, y) \in R_L \Rightarrow y \in C_A] \\ &\quad \vee \forall y[(x, y) \in R_L \Rightarrow y \in C_B])\} \\ C_{L \circ (A+B)} &\supseteq C_{L \circ A + L \circ B} \\ \text{(b)} \quad R_{L \circ (A+B)} &= R_L \cdot R_{A+B} \\ &= R_L \cdot (R_A + R_B) \\ &= R_L \cdot R_A + R_L \cdot R_B \quad (\text{see [11]}) \end{aligned}$$

$$\begin{aligned} &= R_{L \circ A} + R_{L \circ B} \\ &= R_{L \circ A + L \circ B}. \end{aligned}$$

Theorem 8

$$(L_1 \circ L_2) \circ L_3 = L_1 \circ (L_2 \circ L_3).$$

$$\begin{aligned} \text{(a)} \quad R_{(L_1 \circ L_2) \circ L_3} &= R_{L_1} \cdot (R_{L_2} \cdot R_{L_3}) \\ &= (R_{L_1} \cdot R_{L_2}) \cdot R_{L_3} \quad (\text{see [11]}) \\ &= R_{(L_1 \circ L_2) \circ L_3}. \end{aligned}$$

(b) To show that

$$C_{(L_1 \circ L_2) \circ L_3} = C_{L_1 \circ (L_2 \circ L_3)}$$

it is necessary to show that

$$\forall b \forall c[(x, b) \in R_{L_1} \wedge (b, c) \in R_{L_2} \Rightarrow b \in C_{L_2 \circ L_3} \wedge c \in C_{L_3}]$$

is equivalent to

$$\forall b \forall c[(x, b) \in R_{L_1} \wedge (b, c) \in R_{L_2} \Rightarrow b \in C_{L_2} \wedge c \in C_{L_3}].$$

This follows from

$$C_{L_2 \circ L_3} = \{x: x \in C_{L_2} \wedge \forall y(x, y) \in R_{L_2} \Rightarrow y \in C_{L_3}\}.$$

Acknowledgments. Terry Baker, Mike Blasgen, C. Chandrasekaran, Walter Ellis, Stuart Faulk, Mary Forthofer, Rudy Krutar, John McLean, Harlan Mills, Matt Perriens, John Shore, Don Stanat, and Doug Waugh made helpful comments on early drafts of this work. C. Landwehr helped with Example 3. Cliff Jones offered some helpful observations on nondeterminism. Mary Forthofer strengthened my motivation with her thought-provoking concepts on the application of control structures in environments that produce real (useful) programs. Flaviu Cristian and Roy Mad-dux both make useful comments on an earlier version of this work. Harlan Mills's encouragement of this work is deeply appreciated. Typing support by Dee Wilson and Clarissa LaSalle made a real difference.

REFERENCES

1. ACM statement on ADA. *Comm. ACM* 25, 2 (Feb. 1982).
2. de Bruijn, N.G. Private communication.
3. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
4. Forthofer, M.J. *Extending PDL to include a search superstructure*. Tech. Rept. FSD 81-0010, Federal Systems Div., IBM Corp., Bethesda, Md., Apr. 1981.
5. Gries, D. *The Science of Programming*. Springer-Verlag, New York, N.Y., 1981.
6. Linger, R.C., Mills, H.D., and Witt, B.I. *Structured Programming Theory and Practice*. Addison-Wesley, Reading, Mass., (1979).
7. Maddux, R.A. *A study of computer program structure*. Ph.d. dissertation, Dept. Systems Design, Univ. Waterloo, Canada; 1975.
8. Mills, H.D. *Functional semantics for sequential programs*. Proc. IFIP 1980, North-Holland, Amsterdam.
9. Mills, H.D. *The new math of computer programming*. *Comm. ACM* 18, 1 (Jan. 1975), 43-48.
10. Parnas, D.L. *An alternative control structure and its formal definition*. IBM Tech. Rept. TR FSD-81-0012, IBM Corp., Bethesda, Md., 1981.
11. Stanat, D.F., McAllister, D.F. *Discrete Mathematics in Computer Science*. Prentice-Hall, Englewood Cliffs, N.J., 1977.

CR Categories and Subject Descriptors: D.1 [Programming Techniques]; D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs—control structures; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying Reasoning about Programs; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—algebraic approaches to semantics

Received 9/81; revised 11/81; accepted 8/82.