D. P. Pazel

Mathematical Construct for Program Reorganization

Abstract: A mathematical formalism is described through which a program is given a symbolic representation and, with the application of several basic formulas, may be transformed into an equivalent representation giving rise to a reorganized program. Examples are given in which programs are simplified (e.g., code is reduced) or reorganized into a structured form. In effect a mathematics is described that applies to programs in much the same manner as Boolean algebra applies to switching circuits.

Introduction

Structured programming is a style whereby proper programs (programs with one input and one output) are composed of, and only of, the following components:

1. Sequential coding blocks
2. IF-THEN-ELSE decisions
3. DOWHILE (DOUNTIL) iteration blocks.

It is widely known [1] that all proper programs can be written as structured programs. Adherence to the structured programming style leads to the elimination of GOTO statements, which undoubtedly poses the most difficulty to programmers attempting to structure their programs.

This problem posed by structured programming is compounded by other concerns. One wants to avoid needless duplication of code. Use of a subroutine may provide an answer, possibly at a cost to clarity. A slight redesign of the program may solve this problem just as well. However, redesign may not be apparent. Other problems concern how to combine decisions (possibly making two DOWHILES into one) and seeing how decisions influence each other so that some decisions may be transposed for clarity or dropped altogether for simplicity.

A formal method of analysis is described in this paper that is intended to make these problems more transparent and their solutions more accessible. The method strongly conforms to the structured programming approach. However, unlike other approaches to program transformation [2], this method uses an algebraic structure rather than a set of transformation rules.

The next section presents the fundamentals of the method in terms bearing a strong resemblance to Boolean algebra. Increasingly difficult situations (specifically, the DOWHILE and the DOUNTIL) are examined in the following two sections so that all programs fall within the range of the technique. We then present an elementary theorem whose corollary provides a rather powerful programming technique, which is then demonstrated. The final section attempts to put the method in perspective as a tool for programming analysis.

Fundamentals

Proper programs are composed of two elements, namely processing blocks and decisions. The term processing block refers to a self-contained segment of code with one input and one output. By the variable nature of this term, whole programs, subroutines, or pages of code may be called processing blocks, as may any assembler language instruction having one input and one output. For example, a subprogram that computes the discriminant of a quadratic equation in a program that computes the solutions of the equation may be called a processing block. In this paper, processing blocks are represented by the capital letters of the alphabet. The class of processing blocks is denoted by Π. An equivalence relation, $=$, may be induced on Π. Two processing blocks, A and B, have $A = B$ if and only if when A and B operate on some defined set of input states they produce the same set of output states.

The essence of a decision is a truth-valued statement or predicate. Any such statement has two logical values: T, meaning that the statement is true in its context, and F, meaning that the statement is false in its context. This value determines various paths in a program. In this paper, all predicates are represented by the lower case letters of the alphabet. If s is a predicate, $\bar{s}$ represents the negation of s. For predicates s and t, $s \vee t$ represents s OR t in the nonexclusive sense and $s \wedge t$ represents s AND t. The class of predicates is denoted by Γ. The truth tables for these conventions are given in Table 1.

Suppose A, B $\in \Pi$ and $a \in \Gamma$. The statement $A +_a B$ represents the statement IF a, THEN A, ELSE B [Fig.

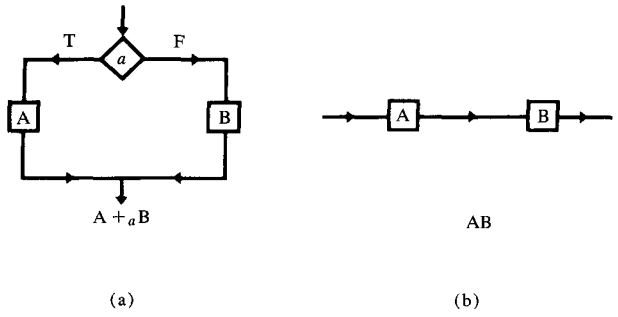


Figure 1 Basic conventions.

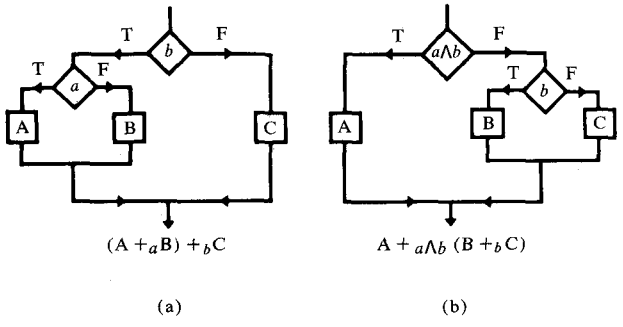


Figure 2 Associative law.

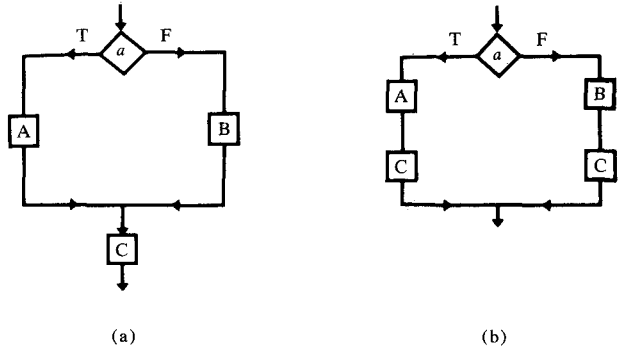


Figure 3 Distributive law.

1(a)]. By convention in any expression using $+_a$, the block processed when the value of a is true is placed to the left of $+_a$. An at most intuitive motivation behind this use of + lies in its OR connotation found in Boolean algebra.

Similarly, the expression AB where A, B $\in \Pi$ represents the sequential processing of A then B (Fig. 1(b)). The convention is to write the blocks from left to right in the order in which they are being processed. Again, the intuitive motivation for this multiplicative notation can be found in Boolean algebra. With these conventions established, elementary relationships can now be derived.

Consider the expression $A +_a B$, where A, B $\in \Pi$ and $a \in \Gamma$. Precisely the same processing would be achieved if B were processed when $\bar{a}$ is true and if A were processed when $\bar{a}$ is false. In short, for A, B $\in \Pi$ and $a \in \Gamma$,

$$A +_a B = B +_{\bar{a}} A. \tag{1}$$

The next expression considered is $(A +_a B) +_b C$, where A, B, C $\in \Pi$ and $a, b \in \Gamma$. This expression, which represents the situation in Fig. 2(a), is an IF within an IF. Note that A is processed when a and b are true. However, given that a is false or b is false, then B would be processed if b were true and C would be processed if b were false. This restatement is symbolically represented by $A +_{a\wedge b} (B +_b C)$. Figure 2(b) illustrates this expression. Thus, it may be stated that for A, B, C $\in \Pi$ and a, $b \in \Gamma$,

$$(A +_a B) +_b C = A +_{a\wedge b} (B +_b C). \tag{2a}$$

A similar argument provides the following analogous result. For A, B, C $\in \Pi$ and $a, b \in \Gamma$,

$$A +_a (B +_b C) = (A +_a B) +_{a\vee b} C. \tag{2b}$$

This expression is stated more for reference as a useful relationship than as an elementary relationship in that it is readily derived from (1) and (2a) in the following manner.

$$\begin{aligned} A +_a (B +_b C) &= (B +_b C) +_{\bar{a}} A \\ &= (C +_{\bar{b}} B) +_{\bar{a}} A \\ &= C +_{\bar{b}\wedge\bar{a}} (B +_{\bar{a}} A) \\ &= C +_{\bar{b}\wedge\bar{a}} (A +_a B) \\ &= (A +_a B) +_{a\vee b} C. \end{aligned}$$

The next elementary relationship is derived from the rather interesting expression $A +_a A$, where A $\in \Pi$ and $a \in \Gamma$. For either truth value of a block A is processed. Thus, a rule of idempotency is introduced. For A $\in \Pi$, $a \in \Gamma$,

$$A +_a A = A. \tag{3}$$

The rules given unfortunately do not all have equivalent counterparts for block multiplication. For example, for A, B $\in \Pi$, AB does not necessarily equal BA. A quick check of the program having A = "VAR = 1" and B = "VAR = 2" bears out this fact. Furthermore, AA and A need not be equal, as seen by taking A = "VAR = VAR + 1."

One may allow in Π the existence of a block that consists of null processing. In flowchart language, this could be represented as a flowline. This block is given the symbol 1. The appropriateness of this convention is shown in the following elementary expression whose verification is immediate. For A $\in \Pi$,

$$A1 = 1A = A. \tag{4}$$

By proceeding along the lines of an algebraic development, the relationships between our fundamental operations are now studied in terms of a distributive law. It is therefore natural to consider the expression $(A +_a B)C$ with $A, B, C \in \Pi$ and $a \in \Gamma$. This simple program is illustrated in Fig. 3(a). If a is true, AC will be processed. Otherwise, BC is processed. This analysis is summarized by the expression $AC +_a BC$ (Fig. 3(b)). This yields the first distributive law. For $A, B, C \in \Pi$ and $a \in \Gamma$,

$$(A +_a B)\, C = AC +_a BC. \tag{5}$$

Interestingly enough such a general statement for distribution from the left does not hold, i.e., $C\,(A +_a B) \neq CA +_a CB$. The fact that C could alter the truth value of a destroys any hope for such a distributive law, and examples are easily generated. However, if C and a are said to be *independent* when the processing of C leaves the truth value of a unaltered, then the following distributive law holds. For $A, B, C \in \Pi$ and $a \in \Gamma$,

$$C\,(A +_a B) = CA +_a CB, \tag{6}$$

provided C and a are independent.

These rules complete the fundamentals. We now present two examples to illustrate how these elementary relationships can simplify and structure code.

The first example concerns the program illustrated in Fig. 4(a). The symbolic representation for this program is $C +_a (C +_b B)$. In accordance with the fundamentals presented, the following reduction proceeds

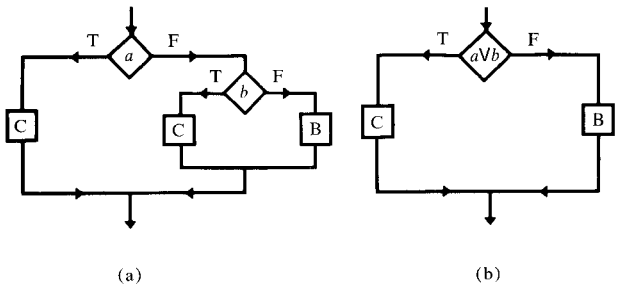
$$\begin{aligned} C +_a (C +_b B) &= (C +_a C) +_{a \vee b} B \\ &= C +_{a \vee b} B \end{aligned}$$

This new expression, illustrated by Fig. 4(b), reflects the removal of a redundant processing block.

A somewhat more complex example is shown in Fig. 5(a). In writing the equation for this or any program, we proceed in a top-down fashion, symbolizing decisions and blocks on every path until all decisions and blocks are represented and the end flow line is reached. Thus, the equation for this example and its reduction are now given.

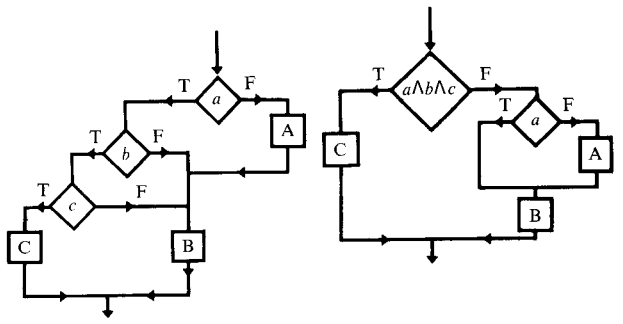
$$\begin{aligned} ((C +_c B) +_b B) +_a AB &= (C +_{c \wedge b} (B +_b B)) +_a AB \\ &= (C +_{c \wedge b} B) +_a AB \\ &= C +_{a \wedge b \wedge c} (B +_a AB) \\ &= C +_{a \wedge b \wedge c} (1B +_a AB) \\ &= C +_{a \wedge b \wedge c} (1 +_a A)\, B. \end{aligned}$$

The new representation for the program is shown in Fig. 5(b). Note that the program is now in structured form with as many unique processing blocks as before and with no block duplications.



(a) (b)

Figure 4 Example of code reduction.



(a) (b)

Figure 5 Example of code structuring.

Table 1 Truth table.

s	$\bar{s}$	s	t	$s \vee t$	$s \wedge t$
T	F	T	T	T	T
F	T	T	F	T	F
		F	T	T	F
		F	F	F	F

Trivial decisions and program breakup

Although the elementary relationships appear to be a good foundation for program simplification and structuring, they are incomplete by themselves. For example, consider the program represented by $AB +_a BC$. Unless certain conditions, such as $A = 1$ with B and a independent or $C = 1$, are known to hold, the fundamental formulas appear useless in combining the two B blocks into one B block. The purpose of this section is to provide a method to achieve this combination.

Consider the program represented by $AB +_a CD$ (Fig. 6(a)). If the processing of blocks A or C did not affect

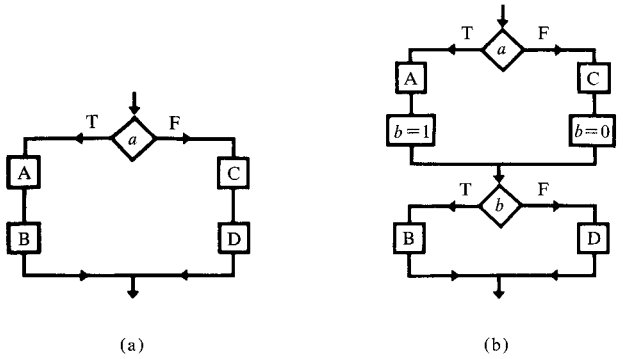


Figure 6 Program breakup by the use of bit settings.

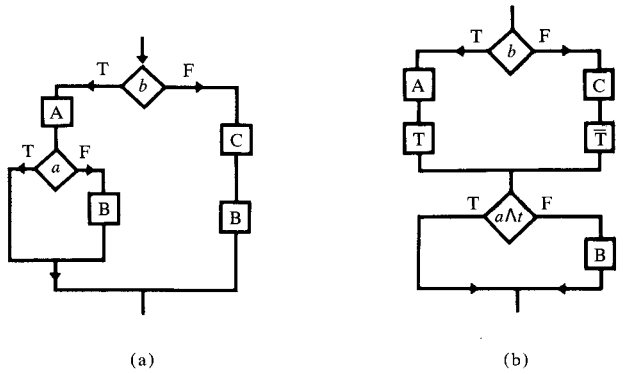


Figure 7 Program breakup with trivial decisions to achieve code reduction.

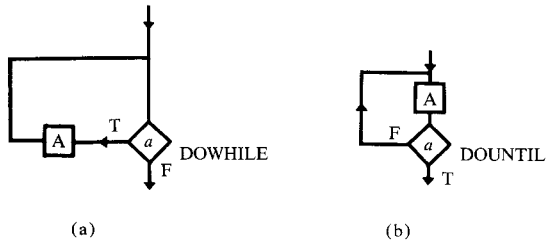


Figure 8 Two basic looping techniques used in structured programming.

the truth value of a, then this program could be broken up so as to have the representation $(A +_a C)\ (B +_a D)$. If, however, the truth value of a could be affected by processing A or C, the same effect could be achieved in the following manner. One may, if a is true, process A and set a bit, $b = 1$. Otherwise, process C and set $b = 0$. Then place a check on b and process B if $b = 1$ and D if $b = 0$ (Fig. 6(b)).

This bit setting and testing provides a method of program breakup that is rather effective in program simplification. To describe this process on a more formal level, a dichotomic variable is introduced into the program. Ideally this variable should be independent of the entire program so as to avoid confusion and to avoid accidental setting (more relevant in the next section). Corresponding to the two settings of this variable are the representations T and $\overline{T}$, respectively. We call T the *positive trivial block* and $\overline{T}$ the *negative trivial block*. Code may now be introduced that depends upon whether T or $\overline{T}$ has been most recently executed. Thus, the decision t is introduced, which, when true, indicates that T has been most recently executed and which, when false, indicates that $\overline{T}$ has been most recently executed. Decision t is called the *trivial decision* with respect to $T(\overline{T})$. Programming situations may introduce several trivial devices. Thus, in such cases, it would be advisable to standardize the notation by subscripting t, T, and $\overline{T}$. A notion similar to this is used in [2].

In line with these thoughts and conventions, the analysis of the situation posed earlier, where a bit was set, yields the following relationship. For A, B, C, D $\in \Pi$ and $a \in \Gamma$,

$$AB +_a CD = (AT +_a C\overline{T})\ (B +_t D). \tag{7}$$

It is not difficult to see that (7) can be generalized for an arbitrary number of blocks in the following manner. For $A_i, B_i \in \Pi$ $(i = 1, \cdots, n)$ and $a \in \Gamma$,

$$A_1 \cdots A_n +_a B_1 \cdots B_n = (A_1\, T +_a B_1\, \overline{T})\ (A_2 +_t B_2) \cdots (A_n +_t B_n), \tag{8}$$

where $T(\overline{T})$ is independent of Ai (Bi) for $1 \leq i \leq n$.

Note that by having $T(\overline{T})$ an independent variable, for $1 \leq i \leq n$, $A_i T = TA_i$ $(B_i\overline{T} = \overline{T}B_i)$. In short, independence gives commutativity.

Equation (8), as it is stated, tends to be a bit misleading for A_1 and B_1 may be null. Because of this, (8) is now rewritten in the following form,

$$A_1 \cdots A_n +_a B_1 \cdots B_n = (T +_a \overline{T})\ (A_1 +_t B_1)\ (A_2 +_t B_2) \cdots (A_n +_t B_n).$$

Recognizing the fact that $T +_a \overline{T}$ is actually a device for storing the truth value of a in an independent location, we adopt the following notation to restate (8) in a more convenient form. Let $\psi(a)$ stand for the storage of the truth value of a, i.e., $T +_a \overline{T}$. Let a' be the decision "Was the value of a true when last stored?" Now for A_i, $B_i \in \Pi$ $(i = 1, \cdots, n)$ and $a \in \Gamma$, (8) becomes

$$A_1 \cdots A_n +_a B_1 \cdots B_n = \psi(a)\ (A_1 +_{a'} B_1)\ (A_2 +_{a'} B_2) \cdots (A_n +_{a'} B_n). \tag{9}$$

For an example of the usage of the concept of trivial decisions, consider the program in Fig. 7(a). It might

be desirable to attempt to eliminate the one occurrence of B. The following conversions of the program succeed in doing just that.

$$\begin{aligned} A\ (1 +_a B) +_b CB &= (AT +_b C\overline{T})\ ((1 +_a B) +_t B) \\ &= (AT +_b C\overline{T})\ (1 +_{a\wedge t} (B +_t B)) \\ &= (AT +_b C\overline{T})\ (1 +_{a\wedge t} B). \end{aligned}$$

The equivalent program is illustrated in Fig. 7(b).

As for the program posed in the beginning of this section, namely to eliminate one of the B blocks in AB $+_a$ BC, the following calculations illustrate the procedure.

$$\begin{aligned} &AB +_a BC \\ &\quad = (ABT +_a B\overline{T})\ (1 +_t C) \\ &\quad = (ATB +_a \overline{T}B)\ (1 +_t C) \quad \text{(independence of } T(\overline{T})) \\ &\quad = (AT +_a \overline{T})\ B\ (1 +_t C). \end{aligned}$$

For a more general and dramatic application of these principles, (9) is now applied to the program whose representation is ACDFG $+_a$ BCEFH.

$$\begin{aligned} &ACDFG +_a BCEFH \\ &\quad = \psi(a) \cdot (A +_{a'} B)\ (C +_{a'} C)\ (D +_{a'} E) \\ &\qquad\qquad \times (F +_{a'} F)\ (G +_{a'} H) \\ &\quad = \psi(a)\ (A +_{a'} B)\ C\ (D +_{a'} E)\ F\ (G +_{a'} H). \end{aligned}$$

Thus, it can be seen that trivial decisions are instrumental in program breakup for program simplification.

Symbolic representation of loops, the DOWHILE and the DOUNTIL

Many programs represent iterative processes. This is achieved through program control via jumps to earlier sections of the program. A statement commonly used to produce this effect is the GOTO. However, in line with structured programming, these effects are to be produced only by the DOWHILE and DOUNTIL statements. In their most elemental forms, each consists of a predicate a and a processing block A. The precise logic of these statements is conveyed by the illustrations in Fig. 8(a) and (b).

An algebraic approach to the DOWHILE, the DOUNTIL, and loops in general may be found in the following considerations. First, if the point where the logic flows back into the program were given an identity such as that of a processing block, then that point would be preserved and reidentified after any equation reduction to a new representation. Second, if the jumps themselves could be given a similar identity, then the loops would be broken and there should be no problem in symbolically representing the program. That approach is developed

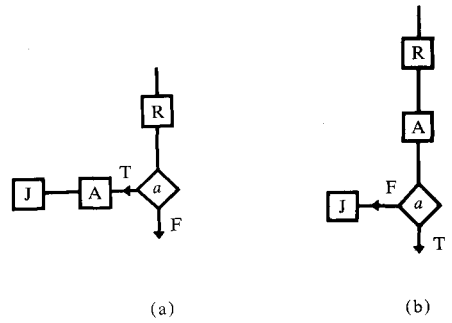


Figure 9 Basic looping techniques are broken apart so as to admit a symbolic representation. J blocks and R blocks in each diagram pair up to form a loop.

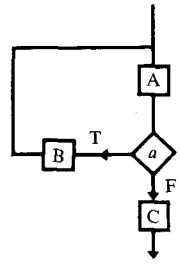


Figure 10 General program configuration, which generates two reorganization laws that lend themselves to the basic looping configurations of structured programming.

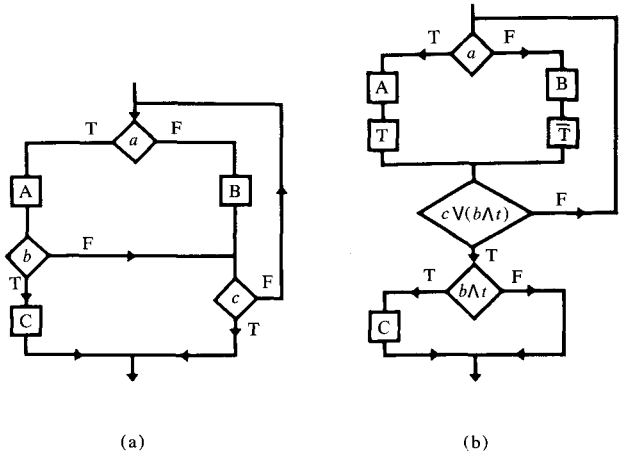
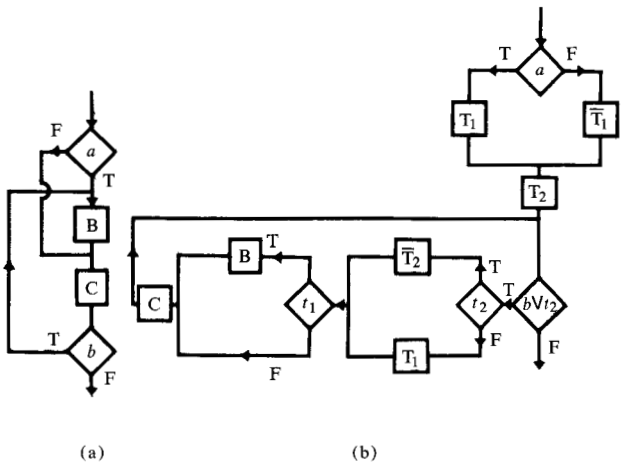


Figure 11 Program reorganization resulting in structured code.

in this section. Thus, for each loop, a "virtual" block R is placed at the point where the logic flows back, and a block J, associated with R (via indexing if necessary to avoid confusion) is placed at the jump. Block R is called a *reception block* and J is called a *jump block*. Blocks R and J are considered to be members of Π.

Consider again the DOWHILE and the DOUNTIL. Figures 9(a) and (b) give alternate illustrations for these logics, which yield the logical expressions R (AJ $+_a$ 1) and RA (1 $+_a$ J) for the two statements, respectively.



(a) (b)

Figure 12 Program reorganization resulting in new structured code that has become somewhat complicated by trivial decisions.

In line with these established conventions and in line with structured programming, the symbolic representation of any program involving loops should be reduced so as to conform to the symbolic forms of the DOWHILE and the DOUNTIL just presented. In order to achieve these forms, several additional rules are now introduced. The first two rules, which are readily verified, concern the logic illustrated in Fig. 10. Both of these rules involve the easily recognized trivial decisions.

In the case of the DOWHILE, for A, B, C $\in \Pi$ and $a \in \Gamma$,

$$\mathrm{RA}\ (\mathrm{BJ} +_a \mathrm{C}) = \mathrm{TR}\ ((\overline{\mathrm{T}} +_t \mathrm{B})\ \mathrm{AJ} +_{a \vee t} \mathrm{C}) \qquad (10)$$

In the case of the DOUNTIL, for A, B, C $\in \Pi$ and $a \in \Gamma$,

$$\mathrm{RA}\ (\mathrm{BJ} +_a \mathrm{C}) = \overline{\mathrm{T}}\mathrm{R}\ (\mathrm{B} +_t \mathrm{T})\ \mathrm{A}\ (\mathrm{C} +_a \mathrm{J}) \qquad (11)$$

One additional rule is now introduced that enhances the applicability of the two rules just given. The objective of this rule is to relocate the position of the reception blocks to bring them into the same logical path of the DOWHILE or DOUNTIL decisions. More precisely, consider $(\mathrm{ARB} +_a \mathrm{CD})\ (\mathrm{E} +_b \mathrm{J})$, where R is a reception block, J is a jump block, A, B, C, D, E $\in \Pi$, and a, b $\in \Gamma$. Clearly R is misplaced for a DOUNTIL. If it could be taken out of the IF-THEN-ELSE, then a DOUNTIL would be possible. The following rule, similar to (7), provides a method by which this extraction can be achieved. For A, B, C, D $\in \Pi$, $a \in \Gamma$, and R a reception block,

$$\mathrm{ARB} +_a \mathrm{CD} = (\mathrm{AT} +_a \mathrm{C}\overline{\mathrm{T}})\ \mathrm{R}\ (\mathrm{B} +_t \mathrm{D}), \qquad (12)$$

where each J associated with R is replaced by TJ.

We now give a few short notes on this rule. First, the rationale behind replacing J by TJ is that on the jump only block B would be executed, as demanded by the original IF-THEN-ELSE. Second, consider the situation when B = D = 1. The rule yields

$$\begin{aligned}\mathrm{AR} +_a \mathrm{C} &= (\mathrm{AT} +_a \mathrm{C}\overline{\mathrm{T}})\ \mathrm{R}\ (1 +_t 1)\\ &= (\mathrm{A} +_a \mathrm{C})\ \mathrm{R},\end{aligned}$$

because as t vanishes, so do T and $\overline{\mathrm{T}}$. Consequently, J is not replaced by TJ.

The first of the two examples to which these concepts are applied concerns the flowchart shown in Fig. 11(a). By placing a reception block R and a jump block J in the appropriate places, the loop is broken, and an equation is readily written for the program. The equation reduction follows

$$\begin{aligned}&\mathrm{R}\ (\mathrm{A}(\mathrm{C} +_b (1 +_c \mathrm{J})) +_a \mathrm{B}\ (1 +_c \mathrm{J}))\\ &= \mathrm{R}\ (\mathrm{AT} +_a \mathrm{B}\overline{\mathrm{T}})\ ((\mathrm{C} +_b (1 +_c \mathrm{J})) +_t (1 +_c \mathrm{J}))\\ &= \mathrm{R}\ (\mathrm{AT} +_a \mathrm{B}\overline{\mathrm{T}})\ (\mathrm{C} +_{b \wedge t} ((1 +_c \mathrm{J}) +_t (1 +_c \mathrm{J}))\\ &= \mathrm{R}\ (\mathrm{AT} +_a \mathrm{B}\overline{\mathrm{T}})\ (\mathrm{C} +_{b \wedge t} (1 +_c \mathrm{J}))\\ &= \mathrm{R}\ (\mathrm{AT} +_a \mathrm{B}\overline{\mathrm{T}})\ ((\mathrm{C} +_{b \wedge t} 1) +_{c \vee (b \wedge t)} \mathrm{J}).\end{aligned}$$

The final equation, a structured representation of the program, is shown in Fig. 11(b).

The second example is particularly good, because as it provides an application of the rules just given, it also illustrates the problems presented by trivial decisions. Indeed, although the program represented by Fig. 12(a) is structured, the reformed program has the drawback of giving trivial decisions more than a trivial amount of attention. To proceed, the loop is broken via reception and jump blocks to give

$$\begin{aligned}&(\mathrm{RB} +_a 1)\ \mathrm{C}\ (\mathrm{J} +_b 1)\\ &\quad = (\mathrm{T}_1 +_a \overline{\mathrm{T}}_1)\ \mathrm{R}\ (\mathrm{B} +_{t_1} 1)\ \mathrm{C}\ (\mathrm{T}_1 \mathrm{J} +_b 1)\\ &\quad = (\mathrm{T}_1 +_a \overline{\mathrm{T}}_1)\ \mathrm{T}_2\ \mathrm{R}\ ((\overline{\mathrm{T}}_2 +_{t_2} \mathrm{T}_1)\\ &\qquad \times (\mathrm{B} +_{t_1} 1)\ \mathrm{CJ} +_{b \vee t_2} 1),\end{aligned}$$

as shown in Fig. 12(b).

Associativity theorem

In the second section, certain associativity rules were considered. Although effective in reorganizing a program, these rules presented the following drawbacks: 1) one of the decisions increases in complexity, and 2) one decision becomes duplicated. The purpose of this section is to give a condition whereby these drawbacks are eliminated. Thus, associativity is made pure, i.e., parentheses are moved, but no decisions change. The following main theorem generalizes the goals.

Associativity theorem Given A, B, C $\in \Pi$ and a, b $\in \Gamma$. Suppose there exists $c \in \Gamma$ such that $b = a \vee c$; then

$$(\mathrm{A} +_a \mathrm{B}) +_b \mathrm{C} = \mathrm{A} +_a (\mathrm{B} +_c \mathrm{C}).$$

Similarly, if there exists $c \in \Gamma$ with $a = b \wedge c$, then

$$A +_a (B +_b C) = (A +_c B) +_b C.$$

Proof If $b = a \vee c$, then

$$(A +_a B) +_b C = (A +_a B) +_{a \vee c} C = A +_a (B +_c C).$$

Similarly, if $a = b \wedge c$, then

$$A +_a (B +_b C) = A +_{b \wedge c} (B +_b C) = (A +_c B) +_b B.$$

Now if a implies b, then $b = b \vee a$. Thus, the associativity theorem yields the following corollary.

Corollary If a implies b, then

$$(A +_a B) +_b C = A +_a (B +_b C).$$

The techniques used to prove these facts are elementary. In fact, none of the machinery developed over the past few sections has been used at all. However, the theorem was presented at this point for an application to an example involving a loop. The fact that a conversion to a DOWHILE is achieved without altering the decisions, but indeed by a transposition of decisions, is significant, because at the outset this transformation may not be seen as obvious.

Consider Fig. 13(a). Assume that a implies b. An application of the corollary gives

$$R\,[(AJ +_a B) +_b 1] = R\,[AJ +_a (B +_b 1)].$$

Figure 13(b) shows the new structured form of the program.

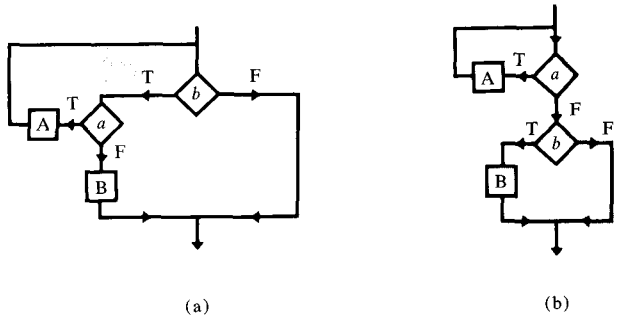


Figure 13 An example illustrating application of the corollary to the associativity theorem. In this example, it is assumed that a implies b.

Conclusion

We have presented informally a foundation for an algebraically motivated mathematics that concerns program structuring, reorganization, and simplification. It contains some rather basic elements intended to cover a wide range of applications.

The point of view taken in this paper has been more practical than theoretical. We have, in fact, used these techniques in a programming environment and found them helpful. However, there have been a few negative aspects. First, the complexity of the equations can increase quite rapidly as larger programs are investigated, adding difficulty and confusion in manipulating them. Second, the method of breaking apart a program with trivial decisions, when done repeatedly, appears to obscure the intent of the program with too many new variables. In fact, the use of trivial decisions to break apart $RA\ (BJ +_a C)$ even seems a distortion of simplicity. (Perhaps this form should be considered a basic programming structure.)

An interesting aspect of this approach is its algebraic organization, in that it gives the appearance, roughly, of a Boolean algebra superimposed over the predicate calculus. The associativity theorem and its corollary seem to indicate that the more information known about the predicates in their relationship to the program, the more power is invested in the algebra as a whole. It is conceivable that substantial development could be achieved along these lines. Because of this aspect, along with the fact that this mathematics does provide a clean method of program reorganization, this mathematics is meaningful and deserves closer study.

Acknowledgment

I especially thank Barry Harding, Terrance Hammond, and John Nilson, whose remarks, interest, and encouragement have been appreciated.

Cited and general references

1. H. D. Mills, "The New Math of Computer Programming," *Comm. ACM* **18,** 43 (January 1975).
2. C. Bohm and G. Jacopini, "Flow-Diagrams, Turing Machines, and Languages with only two Formation Rules," *Comm. ACM* **9,** 366 (May 1966).
3. D. C. Cooper, "Bohm and Jacopini's Reduction of Flow Charts," *Comm. ACM* **10,** 463 (1967).
4. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, New York, 1973.
5. E. W. Dijkstra, "GOTO Statements Considered Harmful," *Comm. ACM* **11,** 147 (1968).
6. D. E. Knuth and R. W. Floyd, "Notes on Avoiding GOTO Statements," *Information Processing Letters* **1** (1971).
7. D. E. Knuth, "Structured Programming with GOTO Statements," *Computing Surveys* **6,** 261 (1974).

Received July 29, 1974; revised February 26, 1975

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.