Functional Programming Analysis

Eldred Nelson

TRW Defense and Space Systems Group

An analysis of several routines from a large real time software system, using recently developed functional programming theory, has shown that the functional capabilities of the routines can be constructed from analysis of the code text. This analysis also showed that the number of distinct functions computed by a program is much smaller than generally appreciated. Many apparent logic paths are not executable, some executable logic paths compute the same function (on different input subsets), and some functions are unnecessarily fragmented by excessive logic tests. With the aid of information derived from the analysis, the routines were restructured into simpler forms, having fewer executable statements and a more visible relationship of code text to functional capabilities. Some of the restructured routines have higher performance—shorter execution time and less primary storage usage. The application of functional programming to the generation of test cases to demonstrate satisfaction of functional requirements, software maintenance, and construction of new programs having visible correspondence to functional requirements is also discussed.

INTRODUCTION

An analysis of several routines from a large real time software system, using recently developed functional programming theory [1],[1] has shown that the functional capabilities of the routines can be constructed from analysis of the code text. With the aid of information derived from this analysis, the routines were restructured into simpler forms, having fewer executable statements and a more visible relationship of code text to functional capabilities. Some of the restructured routines have higher performance—shorter execution time and less primary storage.

The technique of *functional programming analysis* can be applied to essentially any program to determine the functions the program actually computes, the domains of the functions, and the structural elements involved in computing each function. Thus it contributes to "understanding a program." The results of the analysis can be applied in several useful ways, including

- restructuring a program into a simpler form, as was done with the real time system routines,
- improving program performance,
- solving problems in program maintenance, and
- generating test cases.

Additionally, the technique has been adapted to designing new programs to have a visible correspondence of structural elements to functional requirements.

Functional programming theory extends the definition of a program, as a specification of a computable function, to apply to logical structures within the program (executable logic paths, code segments, and branch expressions). It associates each executable logic path with a function defined on the subset of inputs causing execution of the logic path. The further extension of the theory to the code segments and branch expressions composing a logic path provides a basis for analysis of the program text into logical structures from which the functions computed by these structures can be constructed. The theory therefore decomposes the function specified by the program into the set of functions computed by the logic paths and defined on (disjoint) subsets of the input domain.

The application of functional programming analysis to actual routines has shown not only that the functions can be constructed but that the number of distinct func-

[1]The name "functional programming" was suggested by J. R. Brown in 1975 during the initial development of functional programming theory. Since beginning to prepare this paper, the author has become aware of the use of the term to denote a technique for implementing functional design concepts [2] and to denote programming in a new type of programming language based on functional concepts [3].

Address correspondence to Eldred Nelson, Director, Technology Planning and Research, Systems Engineering and Integration Division, TRW Defense and Space Systems Group, One Space Park, Redondo Beach, CA 90278.

The Journal of Systems and Software 2, 225–235 (1981)

0164–1212/82/030225-11$02.75

tions, computed by a specific program, is much smaller than generally appreciated. Many apparent logic paths (called *phantom paths*) are not executable, some executable logic paths compute the same function (on different input subsets), and some functions are fragmented by unnecessary logic tests.

Many *functional programs,* i.e., programs constructed according to functional programming theory, are structured programs; however, functional programming analysis has shown that some structured programs have unnecessarily complicated logical structures, induced by blindly following "rules" for writing structured programs. Such complex structured programs can be simplified to structured functional programs. It has also shown that, in some cases, a structured program does not have the simplest structure.

In the following sections functional programming theory is summarized and its application to analysis of specific routines is described. Other applications of functional programming theory—to the generation of test cases to demonstrate satisfaction of functional requirements and to the construction of new programs having visible correspondence to functional requirements—are also indicated.

FUNCTIONAL PROGRAMMING THEORY

Functional programming theory is based on the definition of a program given in the SEMANOL [4] system: A *program p* specifies a computable function f on the set E of possible inputs. E, the input domain of p, is composed of members E_i, each member being a set of input values for an execution of p:

$$E = \{E_i : i = 1,2, \ldots ,N\}.$$

The input values composing an E_i include all values necessary to the execution of p, including those values, if any, saved from a previous execution of p or provided in a data base accessible to p. Each E_i may be considered to be a set of ordered pairs, $E_i = \{(v_1,a_1), (v_2,a_2), \ldots (v_m,a_m)\}$, associating each input variable v_k with a definite input value a_k. E identifies all distinct computations of the program p:

Each E_i in E corresponds to a possible execution of p.

Each actual execution of p is initiated by an input $E_i \in E$.

The number N of members of the set E is finite, although generally very large, for all programs in which the number of variables and their ranges are finite. The function f is a rule assigning to each E_i a value $f(E_i)$ from a set called the *range* of f.

The case of "nondeterministic" programs [3] may be included by replacing the function f with a relation r allowing execution for a specific input to result in an output value chosen from a set of values specified by r, depending on the environmental conditions existing at execution time. Alternatively, the environmental condition selecting the output value may be interpreted as an additional input.

The finite nature of E also allows execution to be analyzed in finite terms. The case of a program looping until some external event interrupts execution may be analyzed in terms of the finite set of inputs to each traverse of the loop and the finite set of values capable of initiating the interrupt. For those programs containing infinities essential to their description, the theory may be extended using the theory of infinite sets, but for most practical situations such complications are unnecessary.

The function f on E actually computed by a program may not be the function $\hat{f}$ on $\hat{E}$ the program is intended (required) to compute, owing to errors in the program requirements, design, or coding. A "correct" program—i.e., one satisfying its requirements—is therefore a program specifying a computable function f on a domain E such that

$$E = \hat{E}$$

and for all $E_i \in E$,

$$f(E_i) = \hat{f}(e_i).$$

This "functional" view of a program may be extended to program structural elements by considering how a program executes. The execution of p initiated by an input E_i proceeds through a specific code sequence, called an executable logic path L_j. The set of inputs causing execution of the logic path L_j is a subset of E and may be denoted G_j. Execution of p with an input E_i, a member of E but not a member of G_j, causes execution of another logic path L_k, with $k \neq j$. Since all inputs E_i in E cause the execution of some logic path, the association of the inputs into subsets G_j associated with logic paths L_j partitions the input domain E into disjoint subsets.

Since the code sequence for each logic path L_j is itself a program, the logic path L_j specifies a function f_j. The total function f specified by the program p may therefore be represented by a collection of functions f_j, each f_j being defined on a subset G_j of E.

The function f_j on G_j computed by the logic path L_j is called a *functional capability* of p. The function $\hat{f}_j$ on $\hat{G}_j$ the program is required to compute is called a *functional requirement* of p. A "correct" program is accordingly a program for which all functional capabilities

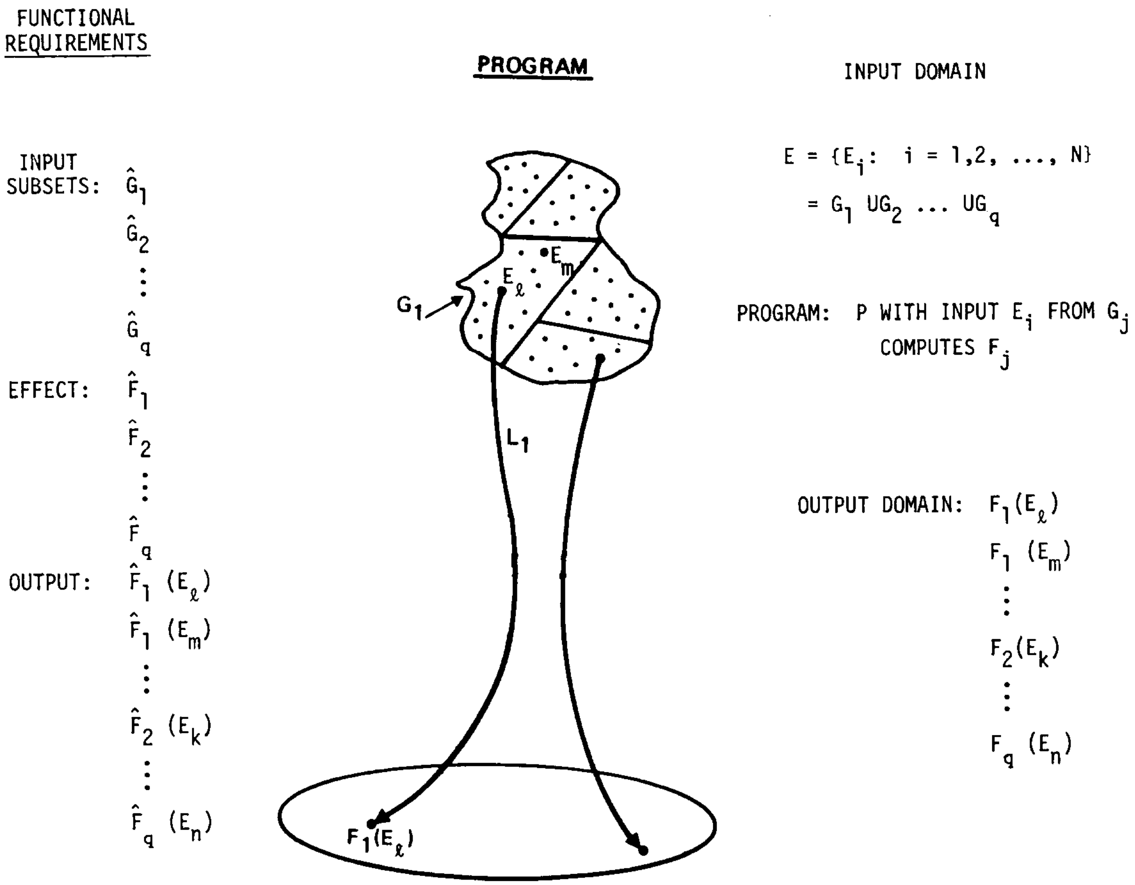

Figure 1. Functional requirements and program logical structures.

equal their corresponding functional requirements; i.e., for all j,

$$G_j = \hat{G}_j, \qquad f_j = f_j.$$

This view of a program is illustrated in Figure 1. The input domain, represented here in two dimensions, is shown partitioned into subsets G_j. The function f_1 specified by logic path L_1 of the program p is shown as a line connecting the input E_l in the subset G_1 with the output value $f_1(E_l)$. The functional requirements $(\hat{G}_j, \hat{f}_j)$, shown at the left of the figure, correspond to the functional capabilities (G_j, f_j), shown on the right.

These functional concepts can be further extended to represent details of execution of logic paths. Given an input $E_i \in G_j$, a program p executes a sequence of statements until it encounters a branch expression (e.g., a boolean expression in an IF statement) specifying a potential change in execution sequence. Evaluation of the branch expression selects the next statement to be executed. A branch expression is therefore a relation on the sets of values over which the variables named in the expression range, partitioning these range sets into subsets associated with the execution sequence selection. Thus an input domain partition G_j is specified by the collective effect of the evaluations of the branch expressions in the logic path L_j associated with the partition G_j. (The situation is complicated in that the values of the variables in a branch expression are not necessarily input values. They may have been computed during an earlier portion of the execution sequence.) The function f_j associated with L_j is specified by the collective effect of the other executable expressions in L_j. The specification of G_j is equivalent to the "weakest precondition" (as defined by Dijkstra [5]) on E such that $(E_i, f_j(E_i))$ holds after L_j is executed; i.e., G_j contains all inputs capable of initiating execution of L_j.

A logic path L_j may therefore be represented in terms of two types of structural components:

an *in-line code segment*, a sequence of executable expressions not containing any branch expressions and having the property that if the first expression in the sequence is executed, all the expressions in the segment will be executed; and a *branch expression,* an expression whose evaluation determines the next expression to be executed.

To make use of these components in functional programming analysis, the symbol S_i is used to represent in-line code segments and the symbol B_j^k to represent branch expressions. The subscript i of S_i takes on integer values denoting the numerical order of the segment in the program. Similarly, the subscript j of B_j^k takes on integer values denoting the numerical order of the branch expression in the program. The superscript k of B_j^k is a variable representing the branch selected by evaluation of the branch expression. k ranges over a set of integers having a lower bound of 0 and an upper bound dependent on the nature of the branch expression. If B_j^k is a boolean expression, the values for k are 0 and 1, with 0 denoting false evaluation of the expression and 1 denoting true evaluation; if B_j^k is the expression in an arithmetic IF statement (e.g., in FORTRAN), the values for k are 0, 1, and 2, with 0 denoting evaluation of the expression as less than 0, 1 denoting evaluation of the expression as equal to 0, and 2 denoting evaluation of the expression as greater than 0; if B_j^k is a case statement, the number of integers in the range of k is equal to the number of cases; etc. (This notation differs slightly from that used in the Functional Programming Report [1].)

GOTO statements are represented by writing GOTO S_i or GOTO B_j^k, denoting transfer of execution to the segment S_i or to the statement containing the branch expression B_j^k.

EXAMPLE OF FUNCTIONAL PROGRAMMING ANALYSIS

Application of functional programming theory to the analysis of programs may be described, using as an example one of the routines of the large real time software system. This routine, called routine A, was written in FORTRAN. Its text is as follows:

```
B1k:      IF(GN.NE.0) GOTO 10;
B2k:      IF(CN.LT.CT) GOTO 5;
S1:       IE = 1
          GOTO 25;
S2:   5   IE = 0;
B3k: 10   IF(CN.LT.TR) GOTO 20;
S3:       IE = 1
          GOTO 25;
S4:  20   IE = 0;
B4k: 25   IF(IE.NE.1) GOTO 40;
S5:       JE = JE + 1
          KI = JD
          KM = 2
          KR = 3
          KG = JA
          KE = JB
          JV = JV + KI + 1
          KG = 1;
S6:  40   RETURN
          END.
```

The text is annotated by listing in a column on the left the symbols representing the segments and branch expressions. (Strictly, the superscripts of B_j^k should be a different symbol for each value of j, since the branch expressions may be evaluated independently; however, by establishing a convention that they are evaluated independently, the complication of writing different superscript symbols may be avoided.)

Functional programming analysis of a program seeks to determine, from the program text, the functions f_j on G_j specified by the program and to associate each function with its corresponding executable logic path L_j. This analysis involves

identifying the input variables and their ranges,

identifying the branch expressions and segments,

constructing a specification for each G_j in terms of branch expression evaluation and any required computations, and

constructing a representation of each f_j in terms of a sequence of segments.

Applied to routine A, this analysis method works as follows: First, the input variables—i.e., the variables that must be assigned values external to the routine in order for the routine to execute—are identified by examining each executable expression and determining which variables, if any, in it must be assigned values in order to evaluate the expression and have not been assigned values by a preceding executable expression. These variables in routine A are easily determined by inspection: GN, CN, CT, TR, JE, JD, JA, JB, and JV.

The ranges of these variables are determined by the FORTRAN convention that undeclared variables beginning with the letters I, J, K, L, M, or N are integer variables and all other undeclared variables are real variables. For GN, CN, CT, and TR, these are the single precision floating point numbers defined for the machine on which execution takes place. For JE, JD, JA, JB, and JV, these are the single precision integers defined for the machine on which execution takes place.

The branch expressions are all boolean expressions in logical IF statements:

B_1^k: GN.NE.0;
B_2^k: CN.LT.CT;
B_3^k: CN.LT.TR;
B_4^k: IE.NE.1;

The (in-line code) segments are

S_1: IE= 1;
S_2: IE = 0;
S_3: IE = 1;
S_4: IE = 0.
S_5: JE = JE + 1
KI = JD
KM = 2
KR = 3
KB = JA
KE = JB
JV = JV + KI + 1
KG = 1;
S_6: RETURN.

In terms of functional programming notation, routine A may be written in a more compact form showing its structure more vividly than the original text:

IF B_1^k GOTO B_3^k
IF B_2^k GOTO S_2
S_1
GOTO B_4^k
S_2
IF B_3^k GOTO S_4
S_3
GOTO B_4^k
S_4
IF B_4^k GOTO S_6
S_5
S_6

From the functional programming notation form of the routine, a diagram (Figure 2) showing the logical structure of the program can be directly prepared. In the diagram the left-hand branch from the lower side of a circle surrounding a branch expression B_j^k represents a false evaluation of B_j^k (i.e., B_j^0) and the right-hand branch represents a true evaluation (i.e., B_j^1). By convention, the flow of execution is downward, so arrows denoting execution flow direction are not used.

It is evident from Figure 2 that routine A has eight potential execution sequences. Represented in functional programming notation, they are

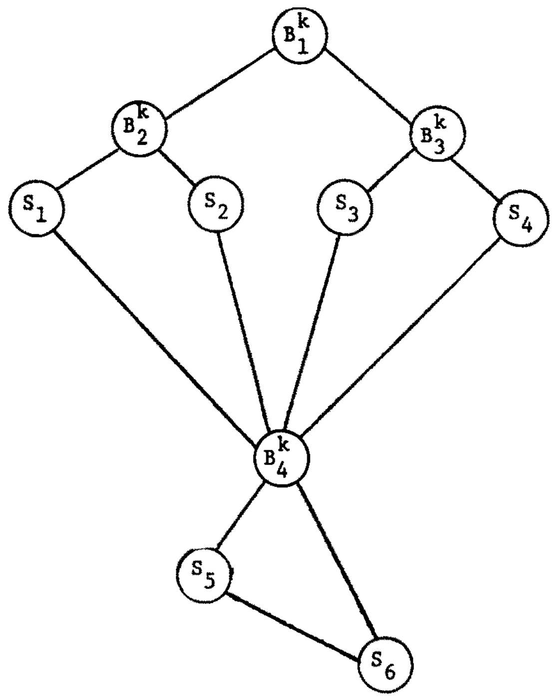

Figure 2. Structure of routine A.

$B_1^0B_2^0S_1B_4^0S_5S_6$,
$B_1^0B_2^0S_1B_4^1S_6$,
$B_1^0B_2^1S_2B_4^0S_5S_6$,
$B_1^0B_2^1S_2B_4^1S_6$,
$B_1^1B_3^0S_3B_4^0S_5S_6$,
$B_1^1B_3^0S_3B_4^1S_6$,
$B_1^1B_3^1S_4B_4^0S_5S_6$,
$B_1^1B_3^1S_4B_4^1S_6$

Phantom Paths

Although there are eight apparent logic paths (the potential execution sequences listed above) through routine A, four of them are not executable. The nonexecutable sequences arise because certain evaluations of the branch expression B_4^k are incompatible with prior assignments of values to the variable IE in segments S_1, S_2, S_3, and S_4. B_4^k branches on whether IE has the value 0 or 1. Since S_1, S_2, S_3, and S_4 set IE to 1, 0, 1, 0, respectively, the sequence $B_1^0B_2^0S_1$ must result in B_4^0, the sequence $B_1^0B_2^1S_2$ must result in B_4^1, the sequence $B_1^1B_3^0S_3$ must result in B_4^0, and the sequence $B_1^1B_3^1S_4$ must result in B_4^1. Thus only the following sequences are executable logic paths:

$B_1^0B_2^0S_1B_4^0S_5S_6$: L_1;
$B_1^0B_2^1S_2B_4^1S_6$: L_2;

$B_1^1 B_3^0 S_3 B_4^0 S_5 S_6$: L_3;
$B_1^1 B_3^1 S_4 B_4^1 S_6$: L_4.

The remaining four code sequences are not executable. They are therefore *phantom paths.*

The phantom paths are not executable because two or more of the B_j^k in the branch expression–segment sequences in the path are, when evaluated, not compatible. They may therefore be identified by finding incompatible branch expression evaluations.

One-half of the apparent logic paths of routine A are executable and one-half are phantom paths. Some programs have more phantom paths than executable logic paths. In such programs, the actual structure of a program can be concealed in a web of phantom paths obscuring the executable logic paths so that they cannot be readily seen by reading the program text.

Because of the existence of phantom paths, the text of a program does not continually cue a programmer on the actual structure of a program as he writes it. He has to keep the picture of the structure entirely in his mind. If the program is more than a few statements long, this may be difficult. The resulting structural confusion may lead to programmer errors. These errors, when made, are not easily detected by scanning the program text. They are found only through extensive and costly debugging and testing.

As a matter of notation, the functional programming symbol L_j is used to represent only executable logic paths and the term "logic path" without any qualifier is used in this paper to denote an executable logic path.

Specification of the Input Domain Partitions G_j

The code sequences for the four executable logic paths all begin with a sequence of two branch expressions: $B_1^0 B_2^0$, $B_1^0 B_2^1$, $B_1^1 B_3^0$, $B_1^1 B_3^1$. These four distinct branch expression sequences are sufficient to define four logic paths. This suggests that the branch expression B_4^k appearing in all the logic paths is not needed. Examination of its nature shows that, in fact, it is not. B_4^k = IE.NE.1 is a test performed on the internal variable IE. This variable is used only in segments S_1, S_2, S_3, S_4, with S_1 and S_3 assigning IE the value 1 and S_2 and S_4 assigning IE the value 0. IE is therefore a flag used to keep track of the logic paths when they converge to a common point in accordance with structured programming rules.

Thus the four input domain partitions may be specified in terms of evaluations of the branch expressions B_1^k, B_2^k, and B_3^k:

G_1: $B_1^0 B_2^0$;
G_2: $B_1^0 B_2^1$;
G_3: $B_1^1 B_3^0$;
G_4: $B_1^1 B_3^1$.

Translated into relations on the actual input variables, the specifications of the G_j become

G_1: GN = 0, CN $\geq$ CT, JE, JD, JA, JB, JV;
G_2: GN = 0, CN $<$ CT;
G_3: GN $\neq$ 0, CN $\geq$ TR, JE, JD, JA, JB, JV;
G_4: GN $\neq$ 0, CN $<$ TR.

For the G_1 and G_3, the branch expression evaluations impose no restrictions on the values of the variables JE, JD, JA, JB, and JV; however, execution of the code specifying f_1 and f_3 can result in overflow in the evaluation of the formulas JE+1 and JV + KI + 1. Thus, the values of JE are restricted to integers not exceeding the maximum single precision integer M minus 1 and the values of JV and KI are restricted by the inequality JV $\leq$ M$-$KI$-$1. This implies the existence of another function f_5, defined on the domain specified by JE$=M$ and JV $>$ M$-$KI$-$1, whose value is the response of the machine to overflow. For G_2 and G_4 these integer variables are not listed in the specifications, because their values are not used in the computations in the corresponding logic paths L_2 and L_4.

Although there are nine input variables, only eight are included in the specifications for G_1, three in the specifications for G_2, eight in the specifications for G_3, and three in the specifications for G_4. Thus each E_i in G_1 is a set of eight input values; each E_i in G_2 is a set of three input values; each E_i in G_3 is a set of eight input values; and each E_i in G_4 is a set of three input values, each input value being a value associated with a specific input variable. The complete input domain E may be constructed by forming the union of its partitions:

$$E = G_1 \cup G_2 \cup G_3 \cup G_4.$$

The partitioning of E by defining different configurations of input variables for each partition G_j effectively defines a structuring of the input data corresponding to the functions f_j computed by the program.

Specification of the Functions f_j

The functions f_j defined by the logic paths L_j (neglecting the overflow function f_5) may be expressed in terms of the sequences of S_i in each logic path:

f_1: S_5S_6;
f_2: S_6;
f_3: S_5S_6;
f_4: S_6

Table 1. Input Domain Partitions and Functions of Routine A

GN = 0		GN ≠ 0	
CN ≥ CT: G_1	CN < CT: G_2	CN ≥ TR: G_3	CN < TR: G_4
f_1 JE = JE + 1 KI = JD KM = 2 KR = 3 KB = JA KE = JB JV = JV + KI + 1 KG = 1 RETURN	f_2 RETURN	f_3 JE = JE + 1 KI = JD KM = 2 KR = 3 Kb = JA KE = JB JV = JV + KI + 1 KG = 1 RETURN	f_4 RETURN

S_1, S_2, S_3, and S_4 do not appear in the function specifications, since they are not used in computing output values. The input domain partitions G_j and their associated functions are shown in Table 1.

The specifications of the f_j in Table 1 are given informally in terms of FORTRAN statements interpretable by programmers, which is considered adequate for an informal understanding of a program. If a formal specification is desired, it can be rewritten in a formal specification language.

It is easily seen from Table 1 that the specification for f_1 is the same as the specification for f_3, and that the specification for f_2 is the same as the specification for f_4; hence, routine A specifies only two distinct functions. Thus, routine A contains functional redundancies and accordingly can be simplified.

Replacing the segment symbols by the code sequences they represent provides the function specifications:

f_1: S_5S_6: JE = JE + 1
KI = JD
KM = 2
KR = 3
KB = JA
KE = JB
JV = JV + KI + 1
KE = 1
RETURN;

f_2: S_6: RETURN.

f_1 computes new values for variables in a data base. f_2 does not update the data base. It just returns to the calling subroutine without computing any values.

Each computed value of f_1 is a combination of the values assigned to the variables JE, KI, KM, KR, KB, KE, JV, and KG at the end of execution of routine A. It is therefore a set of eight output values. Three of the output variables (KM, KR, and KG) are restricted to a single value. The remaining five output variables can individually range over the set of integers fitting within one computer word length, except that overflow considerations may affect the range of JE and JV. (This could lead to defining a third function f_3 having as its output an overflow message.) The range set of f_1 is therefore a set having as its members sets of eight f_1 output values. The mapping from G_1 to this output set is defined by equations corresponding to the assignment statements in S_5.

Restructuring the Analyzed Program

The functional programming analysis of routine A provides sufficient information on the structural properties of routine A to enable the restructuring of the routine in a form containing no phantom paths and no functional redundancies. Since routine A specifies only two functions, it needs only two logic paths and two input domain partitions. The two functions are f_1 and f_2, but the domains on which they are defined must be extended to include G_3 (for f_1) and G_4 (for f_2); i.e., the two input domain partitions G_1' and G_2' are formed by combining G_3 with G_1 and G_4 with G_2.

$$G_1' = G_1 \cup G_3,$$
$$G_2' = G_2 \cup G_4.$$

G_1' and G_2' may be specified in terms of branch expressions:

$$G_1'\text{:}\quad B_1^0B_2^0.\mathrm{OR}.B_1^1B_3^0;$$
$$G_2'\text{:}\quad B_1^0B_2^1.\mathrm{OR}.B_1^1B_3^1.$$

The expressions used in specifying G_1' and G_2' involve an extension of functional programming notation by including the boolean operator .OR. (using FORTRAN-type notation).

The two logic paths may then be written

$$L_1\text{:}\quad (B_1^0B_2^0.\mathrm{OR}.B_1^1B_3^0)S_5S_6;$$
$$L_2\text{:}\quad (B_1^0B_2^1.\mathrm{OR}.B_1^1B_3^1)\ S_6.$$

Restructuring a program containing phantom paths into a functional program containing no phantom paths can be accomplished as in the preceding example by identifying the functions f_j and their composition in terms of the S_i, constructing the logic paths L_j for computing each f_j in terms of the B_j^k and S_i, and converting the functional programming notation into code. For a more detailed discussion of structuring, see Brown and Nelson [1].

A further extension of the notation to include complementation aids the translation of the restructured logic paths into a program. The complement of a boolean branch expression B_j^k will be denoted by $\overline{B_j^k}$ and the complement of an evaluated boolean expression B_1^0 will be denoted by $\overline{B_1^0} = B_1^1$. In terms of complementation, the logic paths may be rewritten

L_1: $(\overline{B_1^1 B_2^1}.\mathrm{OR}.B_1^1\overline{B_3^1})S_5S_6$,

L_2: $(\overline{B_1^1}B_2^1.\mathrm{OR}.B_1^1B_3^1)S_6$.

Since the compound branch expressions in L_1 and L_2 are complements of each other, the restructured program may be written in functional programming notation

IF $(\overline{B_1^k}B_2^k.\mathrm{OR}.B_1^kB_3^k)$ GOTO S_6
S_5
S_6

Substituting the FORTRAN expressions for the functional programming notation furnishes the FORTRAN code for the restructured program:

```
      IF ((GN.EQ.0).AND.(CN.LT.CT).OR.
         (GN.NE.0).AND.(CN.LT.TR)) GOTO 10
      JE + JE + 1
      KI = JD
      KM = 2
      KR = 3
      KB = JA
      KE = JB
      JV = JV + KI + 1
      KG = 1
10    RETURN
      END.
```

The original version of routine A had eight apparent paths, of which four are executable, and 20 executable statements. The restructured version has two logic paths, both executable, and ten executable statements. It therefore has simpler structure and less code. Although the restructuring introduced the complication of a compound boolean expression, that expression defines explicitly the input domain partition G_2.

This version of the program may be called a *functional program,* because its structure is explicitly related to the functions f_j and their input domain partitions G_j. It also has no phantom paths.

ANALYSIS OF OTHER ROUTINES

The application of functional programming analysis to seven other FORTRAN routines produced similar results:

For all the routines analyzed, the functions f_j and their domains G_j were constructed.

Phantom paths were identified in all but the two routines having the simplest structures—i.e., two and three logic paths.

Functional redundancies were identified in three routines.

The routine having the most complex structure (largest number of paths) unnecessarily fragmented some of its functions—i.e., unnecessary logic tests were applied.

The restructured routines contained fewer executable statements.

The results of restructuring are shown in Table 2. An analysis of these routines required solving problems not present in routine A:

Calling of subroutines. The interface and interaction of subroutines with the calling routine can also be analyzed in terms of functions and input domain partitions, extending functional programming to complete programs as well as subroutines.

Loops. The inputs causing different numbers of traverses of a loop involving the same segments and branch expression evaluations may be grouped into a subset G_j and the corresponding set of code sequences may be represented by the logic path symbol L_j. The function f_j specified by a loop path L_j may have a recursive representation. An extension of functional programming notation enables the handling of DO loops.

The analysis of these routines indicates that the number of distinct functions computed by a program (the number of logic paths in the restructured routines) is generally of manageable size, even for relatively com-

Table 2. The Results of Restructuring

	Original routine		Restructured routine	
Routine identifier	apparent paths	executable statements	logic paths	executable statements
B	2	53	2	53
C	3	83	3	83
D	4	16	3	14
E	9	16	3	12
F	18	35	9	27
G	88	29	4	15
H	729	84	21	46

plex programs, and will for such programs be a small fraction of the number of apparent paths. This hypothesis has received independent empirical confirmation by Moranda [6], who used statistical techniques to determine the number of executable logic paths in a program. His technique applied to a program having 141 segments S_i and 70 branch expressions B_j^k and produced a statistical estimate of 40.5 logic paths with a standard deviation of 2.95.

TESTING

Construction of the input domain partitions G_j aids th generation of test cases [7–9]. A test case E_t will belon to a particular G_j and execution of the test case wi cause execution of the logic path L_j computing, as th test output, the function value $f_j(E_t)$. Correct executio of the test case—i.e., $f_j(E_t) = \hat{f}_j(E_t)$—demonstrates satisfaction of the functional requirements $(\hat{G}_j,\hat{f}_j)$ at the point E_t. If the G_j have been constructed, test cases to demonstrate satisfaction of all functional requirements $(\hat{G}_j,\hat{f}_j)$ can be constructed by choosing as test inputs at least one E_i from each G_j. The number of test cases required to assure confidence (in a statistical sense) that $f_j(E_i) = \hat{f}_j(E_i)$ for essentially all E_i in G_j is not very large, owing to the "continuity" of $f_j(E_i)$ within G_j (the same rule is applied to all E_i in G_j to compute $f_j(E_i)$) and to the fact that most software errors in L_j will cause $f_j(E_i)$ to differ significantly from $\hat{f}_j(E_i)$ for almost all E_i in G_j. Construction of the G_j will also permit comparison of each G_j to its corresponding $\hat{G}_j$, verifying by inspection that $G_j = \hat{G}_j$.

Since functional programming analysis identifies all functional capabilities (G_j,f_j) of a program, it enables the comparison of these functional capabilities with the corresponding functional requirements $(\hat{G}_j,\hat{f}_j)$. Such analysis may identify functional capabilities in the program not required by the functional requirements or it may identify functional requirements not implemented in the program. Having determined that the functional capabilities in a program all correspond to functional requirements, one may easily develop test cases covering all functional requirements. Interpretation of test case execution with respect to functional capabilities and functional requirements will aid solving many testing problems.

IMPROVING PERFORMANCE

The restructured programs resulting from functional programming analysis, having fewer executable statements than the programs from which they are derived, generally have improved performance—i.e., shorter execution time and less use of primary storage. The restructured version of routine A was executed with the four test cases used to test the original version of the routine. It executed correctly for all four cases and showed a substantial performance improvement—execution in approximately two-thirds the execution time and use of approximately two-thirds the amount of primary storage. The execution of other restructured routines showed similar but less spectacular performance improvement. The amount of performance improvement, if any, is dependent on the compiler used, for the changes in the branch expressions—e.g., combining branch expressions into compound expressions—may compile into substantially different amounts of code.

Examination of the code segments composing the f_j may disclose additional opportunities for performance improvement—e.g., variables assigned values in L_j but not used in computing the output $f_j(E_i)$.

APPLICATION TO SOFTWARE MAINTENANCE

Software maintenance—solving problems occurring in the operational use of a program and adapting the program to a changing operational environment—is one of the largest components of life-cycle cost. Functional programming analysis can aid software maintenance, because the information it develops—the G_j, f_j, and L_j—aids understanding the program and analyzing problems. For example, a software problem generally can be associated with an input or a set of inputs. From this input or input set it is relatively easy to identify the G_j to which it belongs and therefore the G_j, f_j, L_j combination. The problem then can be analyzed in terms of these components. This should aid identifying the source of the problem and the code to be modified.

A new operational requirement, if it is functional in nature can be expressed in the form $(\hat{G}_j,\hat{f}_j)$. Comparison with the (G_j,f_j) from the functional programming analysis description will show whether the new requirement is a modification of an old requirement or a completely new requirement. With the revised functional requirements, the modifications to the code necessary to implement the revisions are easily designed and test cases to verify the implemented revision are easily constructed.

WRITING FUNCTIONAL PROGRAMS

Since the restructured versions of routines subject to functional programming analysis have several benefits relative to the original versions—fewer executable statements and traceability of code structures to functional requirements—why not obtain these benefits by writing the program initially as a functional program. This can be done by inverting the process by defining the functional requirements first and developing the code segments and branch expressions to implement the functional requirements. Several programs were developed in this manner. The functional programs developed achieved the development goal of visible correspondence of code to functional requirements. Test cases demonstrating satisfaction of requirements were constructed by selecting inputs from the G_j. Examples of functional programs developed from functional requirements are given by Brown and Nelson [1].

RELATIONSHIP TO STRUCTURED PROGRAMMING

The routines analyzed were structured programs, using only certain specified control structures modeling the structured programming control structures of IF THEN ELSE, DO WHILE, etc. in terms of groups of FORTRAN IF statements and GOTO statements. The functional programs developed by restructuring the routines were in all cases except one structured programs. Functional programs may therefore be structured programs but are not necessarily structured programs.

Structured programs have a linear structure, allowing the code text to be read sequentially. This improves structural visibility over older methods of program, because it eliminates the complex, often convoluted branching present in many programs. However, the "return to common point" rule, if employed after each in-line code segment, can result in the formation of phantom paths and require additional branching, for information on the path into a common point is lost on exit from the point.

The fact that most of the functional programs written so far are also structured programs suggests that functional programming concepts and structured programming rules are generally compatible. The case where the functional program was not structured involved branching in the middle of one path to a segment in the middle of another path, creating a third path cross-linking the other two paths. Expressed in functional programming notation, this situation may be represented

L_1: $S_1B_1^0S_2B_2^0S_3S_6$;

L_2: $S_1B_1^0S_2B_2^1S_5S_6$;

L_3: $S_1B_1^1S_4S_5S_6$.

The B_2^1 branch of B_2^k cross links the paths L_1 and L_3 violating structured programming rules. It can be converted into a structured program either by requiring S_4 to return to the common point B_2^k, creating a phantom path $S_1B_1^1S_4B_2^0S_5S_6$, or by repeating the segment S_5 in the code text, either case being a structural complication.

The Jackson Design Method [10] developed concepts similar to those underlying functional programming; however, it stopped short of recognizing phantom paths and the structuring needed to eliminate them.

EXTENSION OF FUNCTIONAL PROGRAMMING ANALYSIS

The initial application of functional programming analysis was to small routines (under 100 executable statements) for which the analysis could be performed manually. Practical application to larger programs will require software tools to perform the large amount of data handling involved in the analysis. The development of useful tools does not appear to be difficult, for many of the needed data-handling functions—the identification of branch expressions, code segments, and variables—are performed by software tools developed for other purposes, and the manual procedure described in this article for analyzing programs can be converted into an algorithm usable in tools. With such tools reasonable sized software systems can be analyzed, constructing the functions they actually perform.

Although functional programming analysis was first applied to analyzing higher-order language programs, it is also applicable to assembly language programs. For such programs, inputs can include storage addresses, the contents of machine registers, and interrupt signals. A collection of eight assembly language routines concerned with interrupt processing for a communications software package was analyzed, constructing the interrupt response functions and their domains.

Functional programming analysis concepts and technique have been extended to the analysis of formal specifications, design specifications written in the design language PDL,[2] and software requirements. There they have proved useful by providing a systematic approach to analyzing the designs. The analysis also identified errors in the designs and requirements [11].

In the application of functional programming analysis to other programs it is expected that situations will be encountered not covered by the rules given in this paper. In such cases, the definition of the computer programs used as the basis of functional programming theory should provide the guidance needed to handle the situation.

The author should like to acknowledge the value of discussions with E. K. Blum, F. Belz, J. R. Brown, H. Hart, M. Lipow, G. Short, and T. Thayer.

[2]PDL (program design language) is a software product of Caine, Farber, and Gordon, Inc.

REFERENCES

1. J. R. Brown and E. C. Nelson, Functional programming, Final Tech. Rep. RADC-TR-78-24, February 1978.
2. M. M. Drossman, Functional software development, Tech. Rep. AFOSR-TR-78-0110 (1978).
3. J. Backus, Can programming be liberated from the von Neumann style?, *Commun. ACM* 21, 613–641 (1978).
4. E. K. Blum, The semantics of programming languages, Part I, Tech. Rep. TRW-SS-69-01, 1969.
5. E. W. Dijkstra, Guarded commands, non-determinancy, and a calculus for the derivation of programs, *Proceedings of International Conference on Reliable Software,* IEEE Cat. No. 75CHO940-7CSR, 1975.
6. P. P. Moranda, Asymptotic Limits to Program Testing, *Proc. COMPSAC* (1978).
7. J. R. Brown and M. Lipow, Testing for software reliability, Teck. Rep. TRW-SS-75-02, 1975.
8. T. Thayer, M. Lipow, and E. C. Nelson, Software reliability study, Tech. Rep. TRW-SS-76-03, 1976.
9. E. C. Nelson, Estimating software reliability from test data, *Microelectron. Reliability* 17, 67–74 (1978).
10. M. Jackson, *Principals of Program Design,* Academic, New York, 1975.
11. J. Logan, Software reliability, application of a reliability model to requirements error analysis, *Proceedings from the Fifth Annual Software Engineering Workshop,* NASA Goddard Space Flight Center, 1980.

Received and accepted 20 December 1981