University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

The Harlan D. Mills Collection | Science Alliance

1975

New Math of Computer Programming

Harlan D. Mills

Follow this and additional works at: https://trace.tennessee.edu/utk_harlan

Part of the Software Engineering Commons

Recommended Citation

Mills, Harlan D., "New Math of Computer Programming" (1975). *The Harlan D. Mills Collection.* https://trace.tennessee.edu/utk_harlan/24

This Article is brought to you for free and open access by the Science Alliance at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in The Harlan D. Mills Collection by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

$$X_1 = 0, \ R_1 = I, \ P_1 = M^*, \tag{17a}$$

$$X_{k+1} = X_k + a_k P_k, \ R_{k+1} = R_k - a_k M P_k, \quad P_{k+1} = M^* R_{k+1} + b_k P_k, \tag{17b}$$

$$a_k = c_k / d_k, \ d_k = | M^* P_k |^2, \quad c_k = | M R_k |^2, \ b_k = c_{k+1}/c_k. \tag{17c}$$

Again if no roundoff errors occur, the algorithm will terminate in m steps, where m is the number of distinct principal (singular) values of M. The matrix X_{m+1} is the pseudoinverse M^{-1} of M. Algorithms (16) and (17) yield the same estimates $X_1, X_2, \ldots, X_{m+1}$ of M^{-1}.

Finally algorithm (15) can be extended to obtain the following routine for computing M^{-1}.

$$X_1 = \hat{X}_1 = 0, \ R_1 = I, \ P_1 = M^*, \ \sigma_1 = 1, \tag{18a}$$

$$X_{k+1} = X_k + a_k P_k, \ R_{k+1} = R_k - a_k M P_k, \quad P_{k+1} = M^* R_{k+1} + b_k P_k, \tag{18b}$$

$$\hat{X}_{k+1} = (X_{k+1} + b_k \sigma_k \hat{X}_k)/\sigma_{k+1}, \ \sigma_{k+1} = 1 + b_k \sigma_k, \tag{18c}$$

$$a_k = c_k / d_k, \ d_k = | P_k |^2, \quad c_k = | R_k |^2, \ b_k = c_{k+1}/c_k. \tag{18d}$$

The algorithm will terminate either where $R_{m+1} = 0$ or where $MP_{m+1} = 0$. If $R_{m+1} = 0$ then $X_{m+1} = M^{-1}$. If $MP_{m+1} = 0$ then $\hat{X}_{m+1} = M^{-1}$. Again m is the number of distinct principal values of M.

Algorithms (16), (17), and (18) for computing M^{-1} have the advantage in that they are simple to execute. Moreover, good estimates of M^{-1} are obtained quickly if the principal values of M are clustered. As is well known, a conjugate gradient algorithm must be programmed carefully in order to avoid unnecessary roundoff errors. It should be noted that there exist computational routines for finding M^{-1} that require fewer multiplications than are required by the algorithms given here. For example, if M is a nonsingular $(n \times n)$-dimensional matrix, the number of multiplications in the algorithms here given may be of the order of n^4 instead of the usual n^3 multiplications required for obtaining the inverse of M.

Received August 1974

The New Math of Computer Programming

Harlan D. Mills
IBM Federal Systems Division

Structured programming has proved to be an important methodology for systematic program design and development. Structured programs are identified as compound function expressions in the algebra of functions. The algebraic properties of these function expressions permit the reformulation (expansion as well as reduction) of a nested subexpression independently of its environment, thus modeling what is known as stepwise program refinement as well as program execution. Finally, structured programming is characterized in terms of the selection and solution of certain elementary equations defined in the algebra of functions. These solutions can be given in general formulas, each involving a single parameter, which display the entire freedom available in creating correct structured programs.

Key Words and Phrases: structured programming, algebra of functions, stepwise refinement, program correctness

CR Categories: 4.6, 5.21, 5.24

In honor of Alston S. Householder

Computer Programming

History

Computer programming as a practical human activity is some 25 years old, a short time for intellectual development. Yet computer programming has already posed the greatest intellectual challenge that mankind

Copyright © 1975, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Invited Address, ACM-Southeast Region Conference, Nashville, Tennessee, April 19, 1974. Author's address: IBM Federal Systems Division, 18100 Frederick Pike, Gaithersburg, MD 20760.

has faced in pure logic and complexity. Never before has man had the services of such logical servants, so remarkable in power, yet so devoid of common sense that instructions given to them must be perfect, and must cover every contingency, for they are carried out faster than the mind can follow.

The practical electronic computer was the invention of some of our best minds in mathematics and engineering [7], e.g. von Neumann, Goldstine, Burks, Bigelow, Williams, Eckert, Mauchly, Atanasoff, Pomerene. Many people from the world's best universities and laboratories came into its development early, in both hardware design and programming, e.g. Wilkes [17], Forrester, Alexander, Forsythe, Rutishauser, Hopper. In the beginning, the emphasis was on numerical computation, and a new mathematics for numerical analysis emerged, spearheaded by the classic studies of von Neumann and Goldstine [16], Householder [10], Wilkinson [18], Henrici [8], et al. Later an additional emphasis developed in symbolic computation, and another new mathematics for symbolic analysis emerged, spearheaded by McCarthy [13], Newell and Simon [15], Minsky [14], et al. The hallmark of numerical computation is iteration and real analysis, and the main conceptual problem is the approximation of iterative algorithms for the reals in floating point numbers. The hallmark of symbolic computation is recursion and combinatorial analysis, and the main conceptual problem is the representation of complex objects in flexible recursive data structures.

The foregoing required computer programming of mathematical processes. But it is only recently that a new mathematics of computer programming itself has begun to emerge, in works of Dijkstra [6], Hoare [9], Wirth [19], et al. In this case, the mathematics models the mental processes of programming—of inventing algorithms suitable for a given computer to meet prescribed logical specifications. Bauer [2], Dijkstra [5], and Knuth [11] have summarized much of this development and its unique characteristics under the term structured programming.

A Mathematical Perspective

We discuss structured programming in mathematical form to illustrate the relevance and power of classical mathematical concepts to simplify and describe programming objects and processes. It is applied mathematics in the classic tradition, providing greater human capability through abstraction, analysis, and interpretation in application to computer programming.

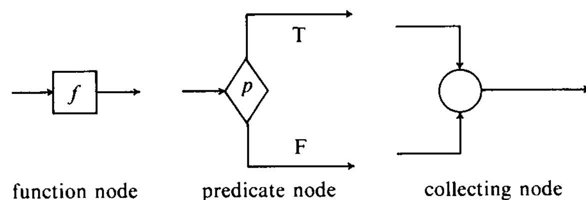
Our principal objective is to model the mental process of structured programming with the selection and solution of certain function equations which arise as a natural abstraction of concrete programming processes. Before these function equations can be abstracted, however, we need to develop the idea of structured programming, and the corollary that structured programs can be viewed as compound function expressions in the algebra of functions. It is the algebraic properties of structured programming that provide its practical power—in the natural nesting of algebraic expressions—and the ability to consider a nested expression independently of its environment in a compound expression.

In illustration, we can all remember from elementary mathematics classes that the problem wasn't simply to get the right answer, but to find the right process for getting the answer. Frequently we got only part credit for a correct answer because we didn't show how we got it. There was a reason. If we do simple mathematical problems by guessing the answers, then when we get to the harder problems we won't be able to guess the answers. That is exactly the role of the new math in computer programming—to go from programming as an instinctive, intuitive process to a more systematic, constructive process that can be taught and shared by intelligent people in a professional activity.

Structured Programming

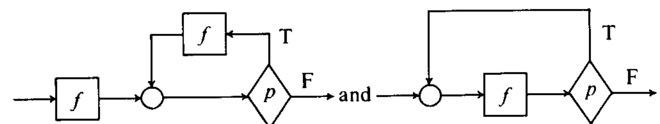
Flowchart Theorems

Flowcharts are graphical rules for defining complex state functions[1] in terms of simpler state functions known to a computing device. More precisely, let X be a finite set of possible states of a computation; a flowchart is an oriented, directed graph with three kinds of nodes:



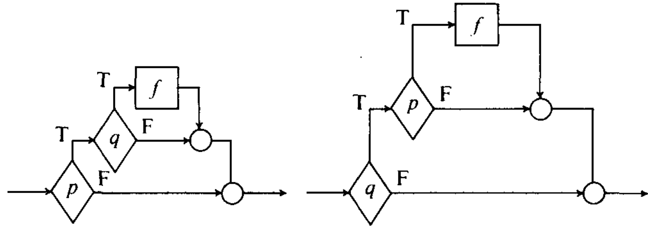
A function node is labeled with a finite state function, say $f \subset X \times X$. A predicate node is labeled with a finite state predicate, say $p \subset X \times \{T, F\}$, and directs control to one of the two out-lines of the node. A collecting node is not labeled, and merely passes control from the two in-lines to the out-line.

Different flowcharts may define the same calculations and same functions, e.g.



[1] A *function* is a set of ordered pairs, say f, with all first members unique. If $(x, y) \in f$ we may write $y = f(x)$ instead, and call x an *argument*, y a *value* of f. The set of all arguments, values is called the *domain*, *range* of f, denoted $D(f)$, $r(f)$ respectively.

define identical calculations. Different flowcharts may define different calculations, but the same function, e.g.

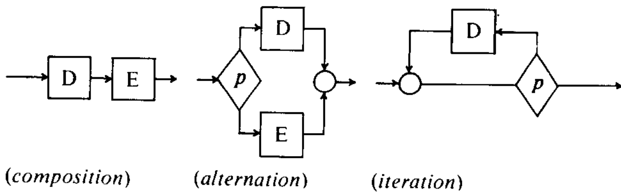


Thus, several levels of flowchart equivalence can be defined, which preserve calculations, function, etc. In particular, Bohm and Jacopini [3], Cooper [4], and others have studied the expressive power of various classes of flowcharts in defining calculations and functions. The principal outcome of these studies is that relatively small, economical classes of flowcharts can define the calculations and functions of the class of all flowcharts, possibly at the expense of extra calculations outside the original description of the state set.

The foregoing motivates a more formal treatment, as follows. Define a class of D-*charts* (D for Dijkstra [5]) over a set of state functions $F = \{f_1, \ldots, f_m\}$ and a set of state predicates $P = \{p_1, \ldots, p_n\}$ as follows:

1. If $f \in F$, then → [f] → is a D-chart

2. If $p \in P$ and → [D] →, → [E] → are D-charts, then

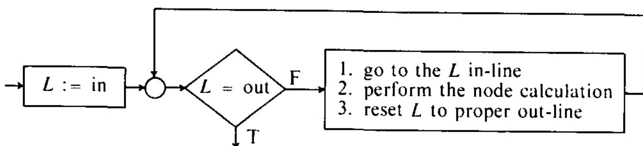


(*composition*) (*alternation*) (*iteration*)

are D-charts.

A Structure Theorem. *Consider any flowchart whose functions form a set F and predicates form a set P. Augment sets F and P with functions and predicates which set and test variables outside the state set of the given flowchart. Then there exists a D-chart in the augmented sets which simulates the calculations of the given flowchart.*

In illustration, following Cooper [4], consider any given flowchart, and label each of its lines uniquely. Then the following flowchart, using a new variable L (for label), will simulate the calculations of the original flowchart.



The operation inside the loop can be expanded into a loop-free D-chart of tests on L, leading to the various nodes of the original flowchart, as a set of nested alternations. In brief, this flowchart shows that, at the expense of setting and testing a single variable L (outside the original state set), the calculations of any flowchart whatsoever can be simulated as a subsequence of the calculations of a D-chart with a single loop.

Bohm and Jacopini [3], Ashcroft and Manna [1], and Kosaraju [12] have sharper results, which preserve more of the structure of the original flowchart. Bohm and Jacopini preserve the loops of the original flowchart, with a more efficient simulation of its calculations. Kosaraju has found a hierarchy of expressive capabilities among several classes of flowcharts. In particular, Kosaraju has discovered the precise conditions under which a D-chart can simulate a given flowchart without augmenting its functions and predicates.

Theorem (Kosaraju [12]). *Consider any flowchart A whose functions form set F, and whose predicates form set P. Then, there exists a D-chart over F and P which preserves the calculations of the given flowchart A if and only if every loop of A has a single exit line.*

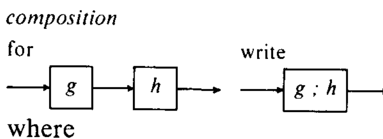
Function Expressions

The algebra of functions inherits function expressions from the algebra of sets, e.g. if g,h are functions, then so are $g \cap h$ (set intersection) and $g - h$ (set difference); of course $g \cup h$ may or may not be a function, but will be a relation in any case.

Basic flowchart programs of common use, such as defined for D-charts, are conveniently represented as additional function expressions. E.g.

composition

for write



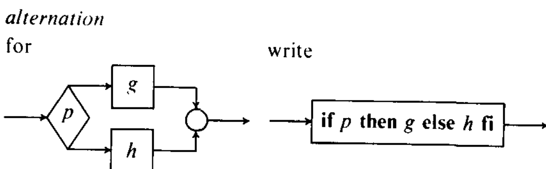
where

(1) $g ; h = \{(x,z) \mid (\exists y)(y \in g(x) \wedge z \in h(y))\}$

(note the operator ; reverses the operands of the ordinary function composition operator $*$, e.g. $g ; h = h * g$).

alternation

for write

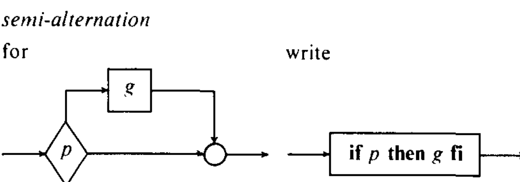


where

(2) **if** p **then** g **else** h **fi** $= \{(x,y) \mid (p(x) \wedge y \in g(x)) \vee (-p(x) \wedge y \in h(x))\}$.

semi-alternation

for write

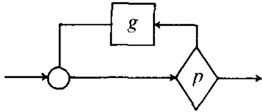


where

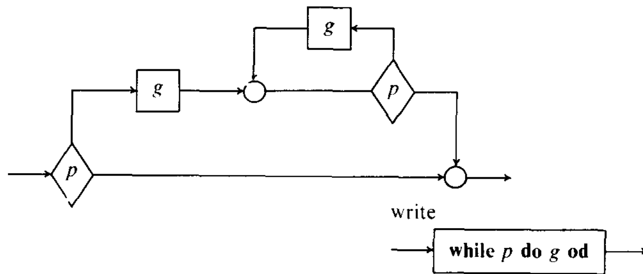
(3) **if** p **then** g **fi** $= \{(x,y) \mid (p(x) \wedge y \in g(x)) \vee (-p(x) \wedge y = x)\}$.

iteration

for



which defines the same calculations as



write

while p **do** g **od**

where

(4) **while** p **do** g **od** = **if** p **then** g ; **while** p **do** g **od fi**.

The iteration expression is defined by recursion in terms of semi-alternation and composition.

As a consequence of these definitions, any D-chart can be represented as a *compound function expression*, and the calculations of any flowchart can be simulated by such an expression.

Additional expression types may be useful and efficient for certain processors, e.g. define

(5) **do** g **until** p **od** = g ; while $-p$ **do** g **od**,

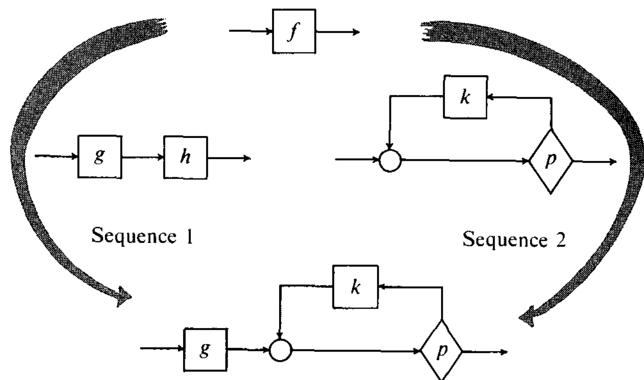
(6) **case** k **of** $g_1, g_2, \cdots, g_n$ **fo** = **if** $k = 1$ **then** g_1 **else**
if $k = 2$ **then** g_2 **else**
...
if $k = n$ **then** g_n **fi** $\cdots$ **fi fi**.

We define a *structured program* to be a compound function expression in any prescribed set of expression types. The D-charts are structured programs in the set of types {composition, alternation, iteration} as defined above.

Stepwise Function Refinement

The powerful properties of structured programming are rooted, finally, in algebraic properties of function expressions. E.g. arithmetic expressions,[2] logic expressions, etc., permit their evaluation, manipulation, etc., a step at a time in innermost subexpressions, independently of their outer environment. We add $2 + 4$ the same way whether we later multiply the result by 9 or divide it by 3, in $9 * (2 + 4)$ or $(2 + 4)/3$. Alternately, a number such as 6 can be expanded as $(2 + 4)$, if useful, or $(2 * 3)$, irrespective of the operations being performed on it. Similarly, function expressions can be formulated and contemplated independently of their environments in more complex compound function expressions.

As noted by Dijkstra [6], Wirth [19], et al., the creative, iterative mental process of structured programming is the *stepwise refinement* of a function into an expression in intermediate functions, until functions available in the computer at hand are reached. Thus, not only is the final expression involved, but also the intermediate mental steps for reaching it are recorded. For example, the sequence of flowcharts labeled 1 and 2 below lead to the same final (structured) program. But sequence 2 does not follow stepwise refinement.



The difference is critical, because sequence 2 contains a mental discontinuity (two, in fact), which requires additional mental processing outside the sequence. In sequence 1, each of the three members are equivalent compound expressions, i.e.

$f = (g ; h) = (g$; **while** p **do** k **od**$)$

But in sequence 2, the first and third members are equivalent, as above, but the middle member is different from either of the others. Thus, from f in sequence 2, by some unrecorded insight, the function called h in sequence 1 is defined as an iteration. This expression equals no other object in sequence 2, and requires that unrecorded insight for validation. Then, at last, this expression is fixed up by putting g in front of it, still needing that unrecorded insight to get g right. When such functions get complex, and many such unrecorded insights need to exist over days, weeks, and months, it is no wonder that programming can be complex and frustrating.

The Correctness of Function Expressions

The verification of correctness of function expressions can proceed with stepwise refinement. In fact they are better practiced jointly than separately and sequentially. Each stage in stepwise refinement identifies a compound expression in intermediate functions, each of which may be later expressed in other functions. These intermediate functions are critical in validating correctness. They serve two roles—first, as functions in expressions being validated, and second as functions by which their replacement expressions are validated.

[2] Exact, not approximate, arithmetic is meant here.

During stepwise refinement, a standard validation procedure can be defined for each expression type. These procedures state what is to be proved—the function description determines how such a proof should be carried out in detail.

THEOREM (CORRECTNESS).

The Correctness of an Alternation Expression. *To prove* $f =$ **if** p **then** g **else** h **fi** *it is necessary and sufficient to show, for every* $(x, y) \in f$, *that either* $p(x) = T$ *and* $y = g(x)$ *or* $p(x) = F$ *and* $y = h(x)$.

The Correctness of a Composition Expression. *To prove* $f = g\ ;h$ *it is necessary and sufficient to show, for every* $(x,y) \in f$, *that* $y = h(g(x))$.

The Correctness of an Iteration Expression. *To prove* $f =$ **while** p **do** g **od** *it is necessary and sufficient to show, for every* $(x,y) \in f$, *that the iteration terminates and that either* $p(x) = T$ *and* $y = f(g(x))$ *or* $p(x) = F$ *and* $y = x$.

The proof of this theorem follows directly from the definitions of (1), (2), (3), and (4).

Function Equations and Their Solutions

The Computation Problem and the Programming Problem

In stepwise refinement, members of a finite set of prescribed function equations arise, one for each expression type, of the forms

(12) $f =$ **if** p **then** g **else** h **fi** (alternation)

(13) $f = g\ ;h$ (composition)

(14) $f =$ **while** p **do** g **od** (iteration)

etc.

When p, g, h are taken as the independent functions, and f as the dependent (unknown) function, these equations represent the *computation problem*; i.e. given a compound function expression, the problem is to evaluate it by stepwise evaluations of innermost expressions.

However, the *programming problem* begins with a function to be expressed, with f as the independent function, and p, g, and h as the dependent (unknown) functions. This motivates the study of these prescribed function equations, with f given, to characterize the solutions in p, g, h. With a little analysis we can write the solutions down directly, and exhibit, thereby, the entire freedom of a programmer in a correct stepwise refinement.

The Alternation Equation

The general minimal solution for the alternation equation can be given in terms of a single parameter, any subfunction (subset) of f, say u. Then (p,g,h) solves the alternation eq. (12), where[3]

$$(15)\quad \begin{aligned} g &= u, \\ h &= f - u, \\ p &= (D(u) \times \{T\}) \cup (D(f - u) \times \{F\}). \end{aligned}$$

Note that $\{g,h\}$ is a partition of f.

The Composition Equation

The general minimal solution for the composition equation can be given in terms of a single parameter, any function, say u, with domain $D(f)$ whose level sets[4] refine the level sets of f. I.e. every level set of u is a subset of some level set of f. Then (g,h) solves the composition eq. (13), where[5]

$$(16)\quad \begin{aligned} g &= u, \\ h &= u^{-1}\ ;f \ \text{ where } (u^{-1} = \{(x,y) \mid (y,x) \in u\}) \end{aligned}$$

Thus, whereas the solution set of the alternation equation has precisely the freedom of a binary partition of the function f, the solution set of the composition equation has the freedom of any system of partitions on the level sets of f, a much richer choice.

The Iteration Equation

The iteration equation is more complex and interesting than the alternation and composition equations. First, whereas any function can be expressed in an alternation or composition, this is not so for an iteration expression; it turns out that an existence condition is required for a solution. Second, whereas all functions p, g, h vary over the solution set in the alternation and composition equations, it turns out that only the function g varies over the solution set in the iteration equation; that is, the predicate p is fixed entirely by f alone. In other words, p is a derivative of f, just as the slope of a differentiable function is a derivative of that function. We call p the *iteration derivative* of f.

Consider the iteration equation, given f, to find (p,g) such that (eq. (14)) $f =$ **while** p **do** g **od.** For the moment, suppose g is restricted to functions for which $D(g) \subset D(f)$; we show below that this involves no loss of generality.

Then we will see that if the existence condition $(x \in D(f) \cap R(f)) \supset f(x) = x$ holds (otherwise there is no solution), the general minimal solution for the iteration equation can be given in terms of a single parameter, a function u which defines any system of trees on the level sets of f in $D(f) - R(f)$, i.e.,[6]

$u = \{(x,y) \mid y \text{ is the parent of } x\}$.

Then (p,g) solves the iteration eq. (14), where

$$(17)\quad \begin{aligned} p &= ((D(f) - R(f)) \times \{T\}) \cup (R(f) \times \{F\}), \\ g &= u \cup (f - D(u) \times R(f)). \end{aligned}$$

In order to see the foregoing, it is easiest to get the

[3] The solution (p, g, h) is minimal, in the sense that, for any other solution (p_0, g_0, h_0), $p \subseteq p_0$, $g \subseteq g_0$, $h \subseteq h_0$. In this case, (p_0, g_0, h_0) must satisfy the additional conditions $\{x \mid p_0(x)\} \cap D(g_0) = D(g)$, $\{x \mid -p_0(x)\} \cap D(h_0) = D(h)$. Nonminimal solutions exist similarly for the other equations, as well.

[4] A level set $D_y(f) = \{x \mid (x, y) \in f\}$, i.e. all arguments with the same value in f. More directly u must satisfy the predicate $D(u) = D(f) \wedge (f(x) \neq f(y) \supset u(x) \neq u(y))$.

[5] In general, u^{-1} will be a relation, not a function, but the composition $u^{-1}\ ;f$ will be a function due to the restriction on u.

[6] More directly, the condition on u is
$u \subset (D(f) - R(f))^2 \wedge (y = u(x) \supset f(y) = f(x)) \wedge u$ acyclic.

formula for p first, then the existence condition, and then the formula for g.

First, for any solution (p,g), p must have value F at every point in $R(f)$, for otherwise the iteration program cannot terminate at that value; conversely, p must have value T at every point in $D(f) - R(f)$, for otherwise the iteration program will not reach a value in $R(f)$. This gives the formula above for p in domain $D(f) \cup R(f)$.

Next, consider any point in $D(f) \cap R(f)$. By the foregoing, p has value F at such a point, and the iteration program never invokes g, but simply exits without altering the state. This gives the existence condition above, i.e. that f must be the identity function on $D(f) \cap R(f)$.

Finally, consider the graph of the state function g in $D(f) \cap R(f)$. It is apparent that the graph of the subset of g in $D(f) - R(f)$ can have no cycles—must be a tree—since otherwise the iteration program would not terminate in such a cycle. It is also apparent that all points of a connected subtree in the graph of g must be in the same level set of f, since the iteration program will terminate at the same value in $R(f)$. Thus the graph of the subset of g contained in $D(f) - R(f)$ must be a system of trees in the level sets of f. Now consider the arcs of the graph of g which originate in $D(f) - R(f)$ and terminate in $R(f)$. The originating points are roots of the trees in $D(f) - R(f)$. Since p is F in $R(f)$, the iteration program terminates with each such arc. Thus, for each such originating point, say x, we must have $g(x) = f(x)$. This gives the formula for g, above, with parameter u, a function defining a system of trees on the level sets of f in $D(f) - R(f)$.

Now we remove the restriction that $D(g) \subset D(f)$ as follows. Suppose $D(g) \not\subset D(f)$; then pick any (x, y) such that $x \in D(g) - D(f), y = g(x)$. If for no $z \in D(f)$ and integer k, $g^k(z) = x$, then (x, y) is superfluous for g and $g - \{(x,y)\}$ is also a solution; otherwise let $g^k(z) = x$, and adjoin $(x,f(z))$ to f, and g remains a solution. In either case the number of elements in $D(g) - D(f)$ is reduced by one; this can be continued until $D(g) \subset D(f)$.

Equations in Compound Function-Expressions

It is direct, but possibly tedious, to extend solutions to function equations in elementary expressions to equations in arbitrary compound expressions of the form f = compound function expression, where no function variable occurs more than once. For each level of nesting an additional parameter is involved, and is effective only within the scope of that nesting. Thus, the parameters of the solution can be associated with the nesting tree of the compound expression.

In particular, the solutions above provide existence predicates on the parameters for each type of function equation, and the formulas for the stepwise refined solutions. These predicates and formulas can be invoked iteratively to describe the set of all solutions to a compound function equation of any complexity. Since there are only a finite number of compound function equations in a fixed number of functions, these formulas permit the explicit formulation of all correct D-chart programs of any size.

References

1. Ashcroft, E., and Manna, Z. The translation of 'go to' programs to 'while' programs. *Information Processing 71*, North-Holland, Amsterdam, 1972, pp. 250–255.
2. Bauer, F.L. "A Philosophy of Programming", U. of London Special Lectures in Computer Science, Oct. 1973: lecture notes published by Math. Inst., Tech. U. Munchen.
3. Bohm, C., and Jacopini, G. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9*, 5 (May 1966), 366–371.
4. Cooper, D.C. Bohm and Jacopini's reduction of flow charts. *Comm. ACM 10*, 8 (Aug. 1967), 463.
5. Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming*. Academic Press, London, 1972.
6. Dijkstra, E.W. A constructive approach to the problem of program correctness. *BIT 8* (1968), 174–186.
7. Goldstine, H.H. *The Computer from Pascal to von Neumann*. Princeton U. Press, 1972.
8. Henrici, P. *Discrete Variable Methods in Ordinary Differential Equations*. Wiley, New York, 1962.
9. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–580, 583.
10. Householder, A.S. *The Theory of Matrices in Numerical Analysis*. Blaisdell, New York, 1964.
11. Knuth, D.E. A review of "structured programming." Stanford Comput. Sci. Dept. Rep. Stan-CS-371, June 1973, 22 pp.
12. Kosaraju, S.R. Analysis of structured programs. *J. Comput. Syst. Sci.* (Dec. 1974) to appear.
13. McCarthy, J. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.), North-Holland, Amsterdam, 1963, pp. 33–70.
14. Minsky, Marvin. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, N.J., 1971.
15. Newell, Allen, and Simon, Herbert. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, N.J., 1971.
16. von Neumann, J., and Goldstine, H.H. Numerical inverting of matrices of high order. *Bull. Amer. Math. Soc. 53* (1947), 1021–1099.
17. Wilkes, M.V. *Automatic Digital Computers*. London, 1956.
18. Wilkinson, J.H. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1965.
19. Wirth, N. *Systematic Programming: An Introduction*. Prentice-Hall, Englewood Cliffs, N.J., 1973.