MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 135

# Mathematical Semantics and Data Flow Programming

by

Paul R. Kosinski

(A paper to be presented at the ACM Symposium on Principles of Programming Languages, Atlanta, January 1976)

This work was supported by a Fellowship from I.B.M. and was done in the Computation Structures Group of Project MAC, MIT.

December 1975

# MATHEMATICAL SEMANTICS and DATA FLOW PROGRAMMING

Paul R. Kosinski

Project MAC
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

and

IBM
Thomas J. Watson Research Center
Yorktown Heights, New York 10598

## Abstract

A Data Flow program [1,2] is a flow-chart like network of operators which compute concurrently, dependent only on the availability of the data which flow along the paths. Each operator has only a local effect, transforming input data to output data. Although operators may exhibit memory and thus not be functional from an input to an output, all operators are functions from input sequences to output sequences. This plus the strong locality of effect allows mathematization of semantics more readily than traditional programming languages which are burdened with omnipresent storage and occasional GOTO's. This paper proves the semantic behavior of some elementary Data Flow programs and proves that certain optimization transformations preserve other behaviors.

## Background

In the past several years, the mathematical specification of programming language semantics has been much investigated. There have been two main lines of attack on this problem, the axiomatic approach taken by Hoare [3], and the functional approach taken by Scott [4] and Strachey.

In the axiomatic approach, each primitive operation in the programming language has assigned to it one or more axioms which formally specify the effect that the operation has upon the state of the abstract machine when that operation is executed. That is, the axioms describe the mathematical relationship between the "before" state and the "after" state. A sequence of operations have an effect which is the composition of the individual relations for the component operations. Thus, given a program together with a putative set of axioms, one can determine by theorem proving (manual or automatic) whether the program does indeed satisfy its axiomatization. Alternatively, one can derive a theorem which describes the program's behavior.

In the functional approach, each primitive operation is assumed to compute a particular function. Thus, a sequence of operations computes the function which is the composition of the component operations' functions. If the operations are performed repeatedly, as in a WHILE loop, the composite function is not so easily determined (in the axiomatic approach, an inductive proof is needed).

Setting up the functional equation corresponding to the loop, one gets

F(X) = IF Test(X) THEN F(Body(X)) ELSE X

where Test is the predicate of the WHILE, Body is the function computed by the body of the loop, and F is the function computed by the loop as a whole. This is a recursive definition, but it is hard to solve because the unknown is the function F. Such equations can be solved in certain circumstances by means of the Y or fixed point operator. Scott's contribution has been to show that there exist lattices called reflexive domains in which the Y operator can always apply to give the unique minimal fixedpoint solution of such equations, and that such domains adequately characterize programming languages.

This approach can be applied to applicative languages with relative ease since such languages are based on the idea of functions and function composition. Unfortunately, applicative languages are seldom used for programming, even LISP has nonapplicative operators such as the GO, SETQ and RPLACD. The effect of such operators is to make the functional characterization of the program depart considerably from the syntactic structure of the program. This occurs for two reasons. First, since some operators, such as assignment (eg. SETQ or worse, RPLACD), change the state of the whole abstract machine, the function corresponding to such an operator must transform states into states. Then, in order to be composable, all operators must transform states whereas the program is written as if most operators transform variables. Second, control flow operators (of which LISP's GO is a mild example) can cause both the conditional and the loop structure of the program to become arbitrarily complicated. Structured Programming, with insistence on a limited disciplined set of control operators (IF-THEN-ELSE and DO-WHILE) prevents the second problem from occurring, that is, one recursive equation corresponds to one loop. The first problem remains however, since most existing languages have state transforming assignment operators.

## Data Flow Semantics

DFPL, a Data Flow Programming Language [2], has the basic mathematical simplicity of applicative languages without most of their drawbacks. Operators in DFPL functionally transform their inputs to their outputs without ever affecting the state of the rest of the program. Since there is no control flow, there is no GOTO; in spite of this, loops may be programmed as well as recursion. Most significant though, is the fact that unlike ordinary applicative languages, programs may exhibit memory behavior, that is, the current output may depend on past inputs as well as the current input. Memory in DFPL is not primitive but is programmed like other nonprimitive operators. Thus its effects are local like those of other operators and it does not permeate the semantics of programs.

A DFPL program is a directed graph whose nodes are operators and whose arcs are data paths. Data in DFPL are pure values, either simple like numbers or compound like arrays or records. There are no addresses in DFPL, although certain operators may be programmed to interpret input values in a manner reminiscent of addresses. An operator "fires" when its required inputs are available on its incoming paths. After a variable amount of time, it sends its outputs on its outgoing paths. It is not necessary that all inputs be present before an operator fires, it depends on the particular operator. Similarly, not all outputs may be produced by a given firing. Synchronous operators fire only when all their inputs are present, and produce their outputs all at once, they are analagous to subroutines. Some operators produce a time sequence of output values from one input

value or conversely, they are analogous to coroutines. The operators in a DFPL program thus operate in parallel with one another subject only to the availability of data on the paths.

An operator may either be primitive or defined. An operator is defined as a network of other operators which are connected by data paths such that certain paths are connected on one end only. These paths are the parameters of the defined operator. A defined operator operates as if its node were replaced by the network which defines it and the parameter paths spliced to the paths which were connected to that node. Thus, recursive operators may be defined.

Sufficient synchronization signals are passed with the data on the paths so that operators do not fire prematurely, and so that the operation of the program as a whole is independent of the timings of the component operators (at least in basic DFPL, full DFPL allows timing dependent programs in order to cope with the real world, but it is not yet possible to mathematize the semantics). Fortunately, the synchronization mechanism is implicit in the mathematization presented here.

There are six primitive operators in DFPL shown in Figure 1. Of these, three are simple in their behavior: the Constant, the Fork and the Primitive computational function (Pcf). This latter is really a whole class of operators including the usual arithmetic, logical and aggregate operators (eg. construct and select). These three operators all have the property that they demand all their inputs to fire, whereupon they produce all their outputs (the constant is a degenerate case having no inputs). Furthermore, each firing is independent of any past history, that is, the operator is a function from current input to current output.

The functional equations for the operators in Figure 1 are thus:

X = C for the Primitive constant,
X = Fx(U,V,W) & Y = Fy(U,V,W)
for the Pcf F,
X = U & Y = U & Z = U for the Fork.

The next most complicated operators are the Switch operators, also shown in Figure 1. These two operators also have the property that each firing is independent of previous firings, but not all inputs/outputs are demanded/produced upon each firing. The Outbound Switch, for example, demands C and U as inputs for each firing, but only one of X, Y and Z receives output. Which one is determined by the value received on input C. The output value is just the value of U. The Inbound Switch operates conversely, only one of the inputs X, Y and Z is accepted upon firing (C is demanded), and its value is always sent out on U.

Since these operators sometimes do not accept/produce inputs/outputs, we can not describe their functional behavior by such simple equations as before (not producing an output is not the same as producing a null output). But we can describe their behavior if we view them as functions from sequences of inputs to sequences of outputs. Now the functional equations for both kinds of Switches are (one origin indexing is assumed):

U* = Inswitch(C*,X*,Y*,Z*) and
X* = Outswitchx(C*,U*) &
Y* = Outswitchy(C*,U*) &
Z* = Outswitchz(C*,U*) where
$X_k = U_j$ if $C_j=1$ & $k=\#\{i\leq j | C_i=1\}$
$Y_k = U_j$ if $C_j=2$ & $k=\#\{i\leq j | C_i=2\}$
$Z_k = U_j$ if $C_j=3$ & $k=\#\{i\leq j | C_i=3\}$

The notation $\#\{i\leq j | C_i=q\}$ means the number of times the length j prefix of C* takes on the value q.

Thus, roughly speaking, the Inbound Switch merges two or more sequences into one sequence the same length as the control sequence. Conversely, the Outbound Switch splits a data sequence into two or more sequences dependent on the values in the control sequence. In all cases, the order of the input sequence(s) is preserved in the output sequence(s).

The most complicated primitive operator is the Loop, shown in Figure 1 also. The Loop provides the DFPL analog of the standard, leading test, WHILE loop of ordinary programming languages. The Loop operator also has the property that it does not accept/produce all of its inputs/outputs each time it fires. Its firing, however, is a two stage process that introduces a "phase shift" of one unit in mapping input sequences to output sequences, thus allowing construction of iterative loops and even an analog to memory or storage in conventional languages.

The four paths connecting to the Loop in Figure 1 can be characterized as follows: X is the initialization value, Y is the current iteration value, Z is the feedback value which becomes current on the next iteration, and C is the control value which tells the Loop whether to stop or take another iteration. Although other Loops can be imagined, such as one having a final output value, they can all be programmed from this minimal Loop plus the primitives above. The precise functional equation for this Loop is:

$Y^* = Loop(X^*,C^*,Z^*)$ where
$Y_1 = X_1$
$Y_k = Z_{k-1}$ if $C_{k-1}=1$ & $k>1$
$Y_k = X_{j+1}$ if $C_{k-1}=0$ & $k>1$ &
$j=\#\{i<k \mid C_i=0\}$

Viewed over "time" (the columns), the Loop operates as follows (the value carried on a path appears under its name if appropriate).

| | | | | | | |
|---|---|---|---|---|---|---|
| $X_1$ | .. | .. | .. | $X_2$ | .. | .. |
| .. | $Y_1$ | $Y_2$ | $Y_3$ | .. | $Y_4$ | $Y_6$ |
| .. | $=X_1$ | $=Z_1$ | $=Z_2$ | .. | $=X_2$ | $=Z_4$ |
| .. | $Z_1$ | $Z_2$ | $Z_3$ | .. | $Z_4$ | $Z_5$ |
| .. | $C_1$ | $C_2$ | $C_3$ | .. | $C_4$ | $C_5$ |
| .. | =1 | =1 | =0 | .. | =1 | =1 |

Now the first three operators can be recast as functions from sequences of inputs to sequences of outputs:

$X_1 = C$ for the Primitive constant
$X_i = Fx(U_i,V_i,W_i)$ &
$Y_i = Fy(U_i,V_i,W_i)$ $\forall i$ for the Pcf F,
$X_i = U_i$ & $Y_i = U_i$ & $Z_i = U_i$ $\forall i$.
for the Fork.

A synchronous operator S is defined as one whose function is such that $Y_j = S(X^*_j)$ where $X^*_j$ = first j elements of $X^*$, ie. there is one output for each input but that output may depend on past inputs also.

This property of synchronous operators allows us to avoid the tedium of using a separate index for the sequence of values on each data path. All paths in a subnetwork of synchronous operators may share the same sequence index since that subnetwork behaves like a single synchronous operator. In general, any operator constructed entirely out of synchronous operators is itself synchronous and the Fork and all Pcf operators are synchronous.

All primitive operators are causal in the sense that an output cannot be affected by future inputs, that is, once an output is produced, it cannot be changed. More precisely, if $Y^* = F(X^*)$ & $Y^*_k = F(X^*_i)$ & $Y^*_\ell = F(X^*_j)$ & $j \geq i$ then $\ell \geq k$.

Optimization

One can prove that natural adaptations of optimization transformations [5] preserve the functionality of certain DFPL programs.

For example, in Figure 2 we see the application of common subexpression elimination. The Before and After program compute the same function for any operator F.

Referring to the "Before" operator definition in Figure 2, we see that $\forall i$: $X_i = X'_i = X''_i$ & $Y_i = Y'_i = Y''_i$ & $Z_i = Z'_i = Z''_i$ by the definition of the Fork operator. Hence, $X^* = X'^* = X''^*$, $Y^* = Y'^* = Y''^*$ and $Z^* = Z'^* = Z''^*$, so $V^* = Fv(X^*,Y^*,Z^*)$, $W^* = FW(X^*,Y^*,Z^*)$, $V'^* = Fv(X^*,Y^*,Z^*)$ and $W'^* = Fw(X^*,Y^*,Z^*)$. Therefore, $V^* = V'^*$ and $W^* = W'^*$. By similar reasoning, in the "After" operator definition of Figure 2, $V^* = V'^* = Fv(X^*,Y^*,Z^*)$ and $W^* = W'^* = Fw(X^*,Y^*,Z^*)$. Thus, the two operators are equivalent for any operator F.

Since Forks have such simple functional properties, we will henceforth omit them as explicit operators in our proofs and just label all paths connected to a Fork with the same symbol.

In Figure 3, we see the application of "hoisting", that is, moving a computation out of a conditional expression. The operator F is moved to the front of the conditional, and the operator G is moved to the rear. For this optimization to apply, it is sufficient for F and G to be simple functions of their inputs (eg. Pcf's), that is $\forall i$: $H_i = F(A_i,B_i)$ & $Z'_i = G(L_i)$.

To prove this optimization, we shall assume that D and E are synchronous operators and that A, B, C and M are mutually synchronized input paths so that we can use the same index for all of them. If these assumptions were not valid, the network would hang up. The proof consists of three parts, first show that R = R' & S = S', second show V = V' & W = W' using the obvious result that U = U' & T = T', and third show that Z = Z'. We will prove the first part in fair detail: the second part is obvious and the third is just like the first.

1. $H_j = F(A_j,B_j)$ by assumption.
2. $R'_k = H_j$ if $C_j=1$ & $k=\#\{i\leq j | C_i=1\}$
   by definition of Outswitch.
3. $R'_k = F(A_j,B_j)$ if $C_j=1$ & $k=\#\{i\leq j | C_i=1\}$
   by 1 and 2 above.
4. $P_k = B_j$ if $C_j=1$ & $k=\#\{i\leq j | C_i=1\}$
   $N_k = A_j$ if $C_j=1$ & $k=\#\{i\leq j | C_i=1\}$
   both by definition of Outswitch.
5. $R_k = F(A_j,B_j)$ if $C_j=1$ & $k=\#\{i\ j | C_i=1\}$
   by assumption for F and 4 above.
6. $R_k = R'_k$ QED.

Similarly, we can prove $S_k = S'_k$, thus concluding the first part of the proof that hoisting preserves the semantics. The first and third parts of this proof stand as separate theorems in themselves. They would not often be used however, because unbalanced Switch operators (ie. an Inswitch without an Outswitch or vice-versa) would rarely be used in programs.

Memory

The most interesting kind of DFPL operator is one which behaves like a memory cell. A trivial kind of memory cell, which serves as the building block for fancier ones, is shown in Figure 4. It is just a holding station, that is, the output is what the input was on the previous firing. More precisely, it can be shown to satisfy the following equations:

$Y_1 = Q$ & $Y_i = X_{i-1}$ $\forall i>1$ .

The proof is straightforward:

1. $W_1 = Q$
   by definition of the Primitive constant.
2. $Y_1 = W_1$ by definition of the Loop.
3. $Y_k = X_{k-1}$ if $Z_{k-1}=1$
   by definition of the Loop operator.
4. $Z_k = True(X_k) = 1$ $\forall k$
   by definition of the Pcf True.
5. $Y_1 = Q$ by 1 and 2 above.
6. $Y_k = X_{k-1}$ $\forall k>1$ by 3 and 4 above, QED.

A fancier memory cell is shown in Figure 5. When a 0 value is presented on the control path C, the current contents is read out on path Y, when a 1 value is presented on C and a data value is presented on the input path X, the cell is updated to contain that new value. The cell has an initial contents of Q. Viewed over time, the Mem operator behaves as follows:

| $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
|---|---|---|---|---|---|---|
| =0 | =0 | =1 | =0 | =1 | =1 | =0 |
| $Y_1$ | $Y_2$ | .. | $Y_3$ | .. | .. | $Y_4$ |
| =Q | =Q | .. | =R | .. | .. | =T |
| .. | .. | $X_1$ | .. | $X_2$ | $X_3$ | .. |
| .. | .. | =R | .. | =S | =T | .. |

The precise formulation of this behavior may be proved to be:

$Y_j = Q$ if $\forall i \leq j: C_i=0$

$Y_j = X_k$ if $C_{j+k}=0$ & $k=\#\{i < j+k | C_i=1\}$

The proof of this follows:

1. $A_1 = Q$ by definition of Hold.
2. $A_\ell = B_{\ell-1}$ $\forall \ell>1$ by definition of Hold.
3. $A_\ell = Y_j$ if $C_\ell=0$ & $j=\#\{i\leq\ell | C_i=0\}$ by definition of Outswitch.
4. $A_\ell = W_k$ if $C_\ell=1$ & $k=\#\{i\leq\ell | C_i=1\}$ by definition of Outswitch.
5. $B_\ell = Y_j$ if $C_\ell=0$ & $j=\#\{i\leq\ell | C_i=0\}$ by definition of Inswitch.
6. $B_\ell = Z_k$ if $C_\ell=1$ & $k=\#\{i\leq\ell | C_i=1\}$ by definition of Inswitch.
7. $Z_k = X_k$ by definition of ⋮ operator.
8. $B_\ell = B_{\ell-1}$ if $C_\ell=0$ by 2, 3 and 5 above.
9. $A_{\ell+1} = A_\ell$ if $C_\ell=0$ by 8 and 2 above.
10. $Y_j = Q$ if $\forall i\leq j: C_i=0$ by induction on 1, 3 and 9 above.
11. $Y_j = B_{j+k}$ if $C_{j+k}=0$ & $j=\#\{i\leq j+k | C_i=0\}$ by 5 above.
12. $B_{j+k} = B_{j+k-m}$ if $\forall 0<i\leq m: C_{j+k-i}=0$ by induction on 8 above.
13. $Y_j = B_{j+k-m}$ if $[\forall 0<i\leq m: C_{j+k-i}=0]$ & $j=\#\{i\leq j+k | C_i=0\}$ by 11 and 12 above.
14. $B_{j+k-m} = X_k$ if $C_{j+k-m}=1$ & $k=\#\{i\leq j+k | C_i=1\}$ by 6 and 7 above.
15. $Y_j = X_k$ if $C_{j+k-m}=1$ & $[\forall 0$ i m: $C_{j+k-i}=0]$ & $j=\#\{i\leq j+k | C_i=0\}$ & $k=\#\{i\leq j+k | C_i=1\}$ by 13 and 14 above.
16. $Y_j = X_k$ if $C_{j+k}=0$ & $j=\#\{i\leq j+k | C_i=0\}$ & $k=\#\{i\leq j+k | C_i=1\}$ from 15 above, by simplifying the if condition, making use of the fact that m is arbitrary in the range 1 to j+k-1.
17. $Y_j = X_k$ if $C_{j+k}=0$ & $k=\#\{i<j+k | C_i=1\}$ from 16 above, since $C_{j+k}=0$ & $k=\#\{i\leq j+k | C_i=1\}$ implies that $k=\#\{i<j+k | C_i=1\}$ & $j=\#\{i\leq j+k | C_i=0\}$.

Steps 10 and 17 above are the desired results for the behavior of the memory cell.

More complicated memories may be programmed by substituting other operators for the Fork and ⋮ operators in Figure 4. For example, by replacing the Fork by a Dequeue operator, and the ⋮ by an Enqueue, a queue memory results. To program a random access memory, another input path, to carry the "address", must be added, as well as replacing the operators.

## Latticework

To make our domains and codomains of data value sequences into lattices, we have to define a partial order on them. Following G. Kahn [6], we say that a sequence A* is "bigger" than a sequence B* if and only if B* is a prefix of A*. The set of sequences (including countably infinite ones) form a lattice under this partial order. The "bottom" of this lattice is the empty sequence. This lattice does not encompass the Scott notion of value approximation, that requires further investigation.

The operator obtained by connecting the output of the Hold operator to a two way Fork, connecting one Fork output back to the Hold input, and making the second

Fork output the output parameter of the defined operator, is the Repeating constant operator. It is characterized by the equation: $X_i = Q \quad \forall i$. That it satisfies this equation can be proved inductively as above. Another way of proving it is to use the lattice fixed-point approach. To do this, we note that our earlier notion of causality exactly corresponds to monotonicity in the lattice. We make the further assumption of continuity, which corresponds to the reasonable assumption that an operator will produce output after a finite sequence of inputs or not at all. Then, referring to our previous description of the Hold operator as a function which transforms any input sequence to an output sequence which is the initial constant prefixed to that input sequence, we see that the minimal fixed-point of this function is the infinite sequence of that constant value. A more detailed proof of an almost identical situation appears in G. Kahn [6].

DFPL currently does not allow operator valued data and thus does not require the existence of Scotts reflexive domains. In spite of this, DFPL allows iterative and recursive programs, both in the practical and mathematical senses. It is hoped that DFPL can be extended to allow operator valued data in the near future, and that this extension can be mathematized with Scott's techniques.

## Conclusions

We have shown that it is possible to develop a mathematical semantics of DFPL in terms of functions from sequences of inputs to sequences of outputs. This mathematization is not complicated by the omnipresence of memory, because memory is local like all other operators, nor by the presence of control flow, which leads to "continuations". The necessity for dealing with input and output sequences is not all bad: many programs (such as database systems) are inherently non-terminating, and cannot be reasonable viewed as simple functions from an input to an output.

As it currently stands, DFPL has a primitive operator which is indeterminate, or timing dependent, in its operation. It would be extremely desirable if it could be characterized as a mathematical function also. To do this would probably require redefining the functions to take datum/time pairs as values, thus complicating the entire system of axioms, theorems and proofs. The theorems of particular interest in this new system would be those which show that certain defined operators, although indeterminate in their internal operation, are completely determinate when considered as atomic operators. Then, those parts of a program which have to be indeterminate in order to deal with the outside world could be so, whereas other parts of the program could be determinate and thus simpler to analyze.

This duality of determinate operators and indeterminate operators suggests the need for a convenient transformation between them. If DFPL programs are viewed as an algebra, then morphisms between such algebras might be defined. In fact, the process of compiling one DFPL program into another (with simpler operators) can be analyzed as a particular morphism.

Hopefully, the approaches set forth in this paper will yield a practical applicability of mathematical semantics to more realistic programs than heretofore possible.

## References

1. J.B. Dennis, "First Version of a Data Flow Procedure Language". MIT Project MAC, Computation Structures Group, Memo 93 (1973).

2. P.R. Kosinski, "A Data Flow Programming Language", IBM Research Report RC 4264 (March 1973).
3. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", Comm ACM 12, pp 576-583 (October 1969).
4. D. Scott, "Outline of a Mathematical Theory of Computation", Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems, pp 169-176 (1970).
5. F.E. Allen and J. Cocke, "A Catalog of Optimizing Transformations", IBM Research Report RC 3548 (September 1971).
6. G. Kahn, "A Preliminary Theory for Parallel Programs", IRIA Laboratory Report 6 (January 1973).

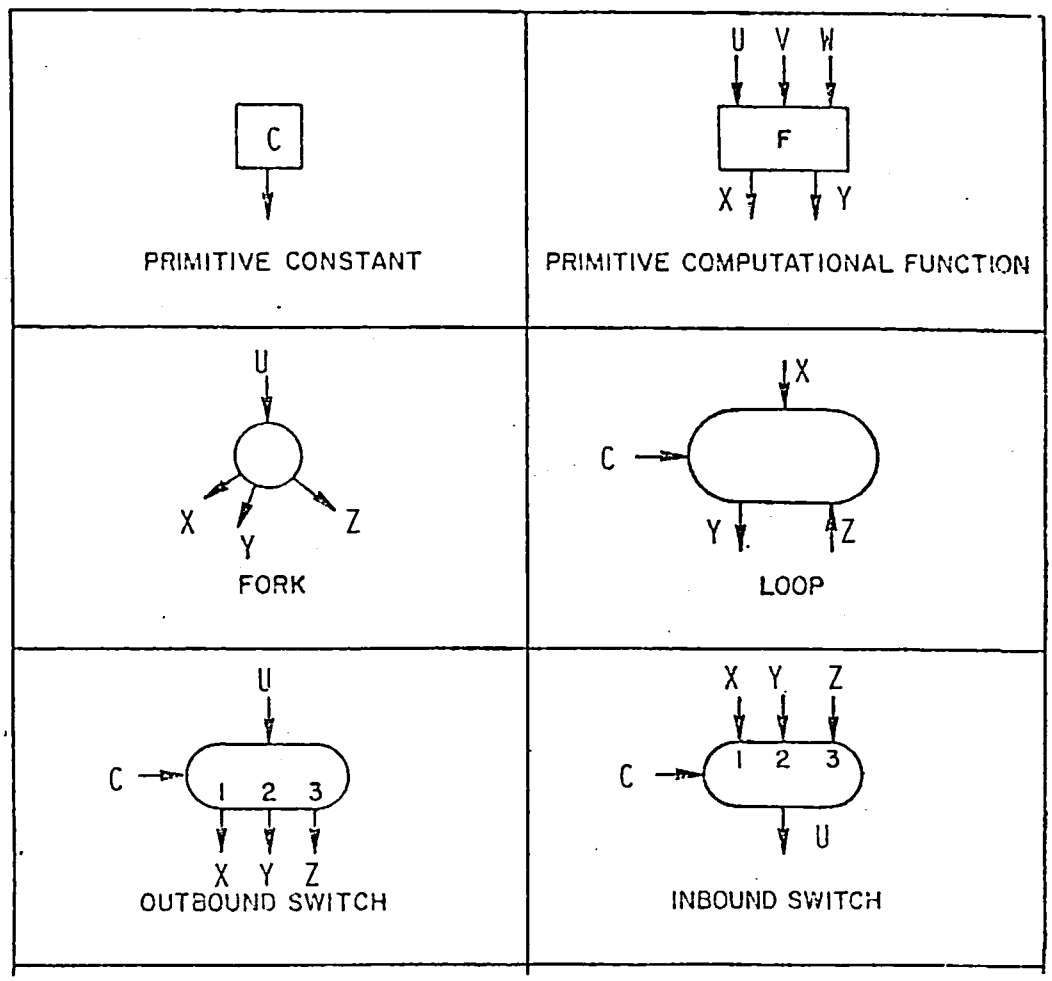
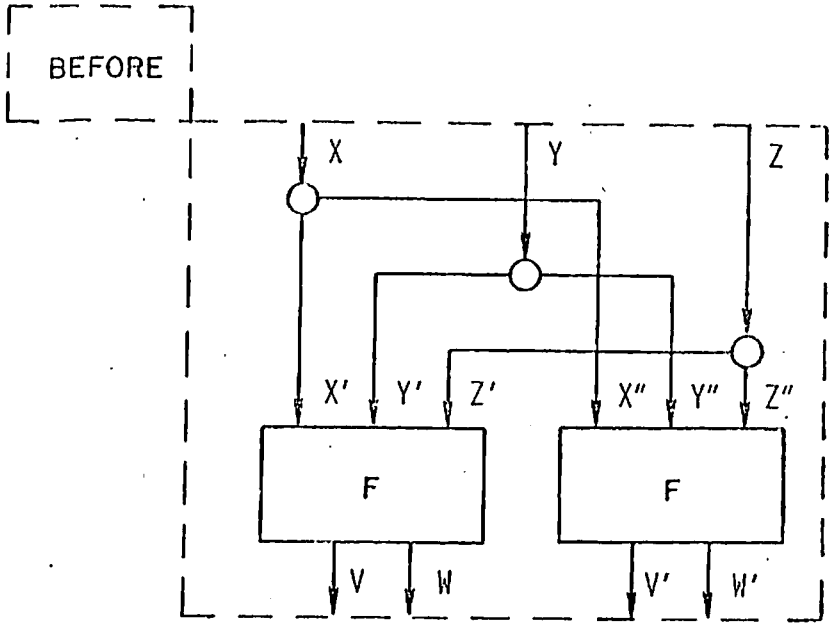



FIGURE 1



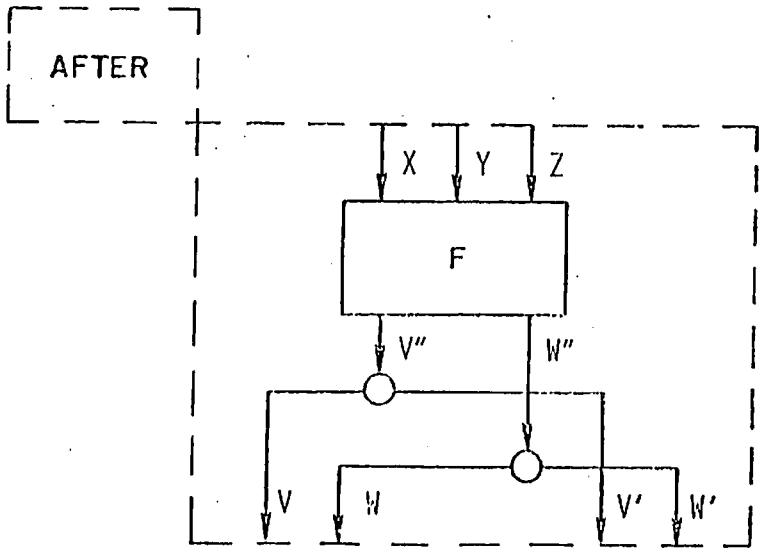
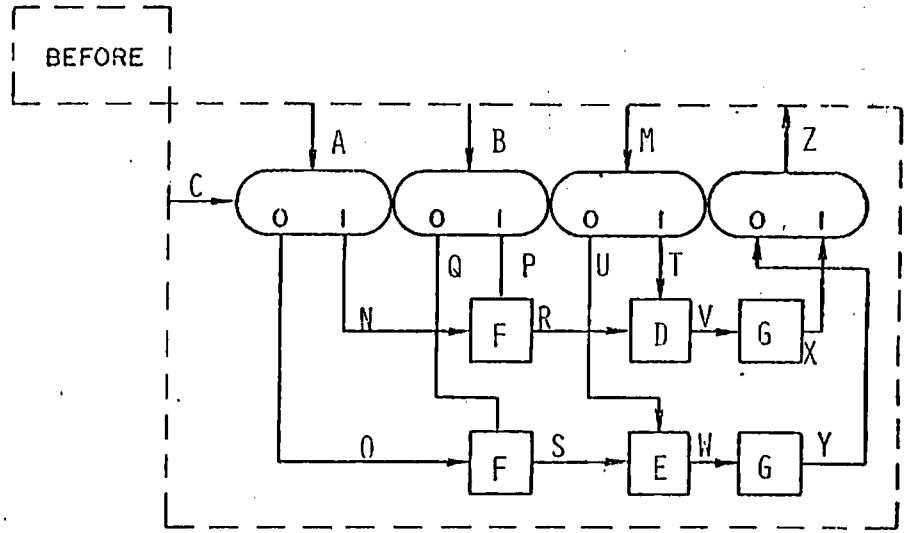




FIGURE 2



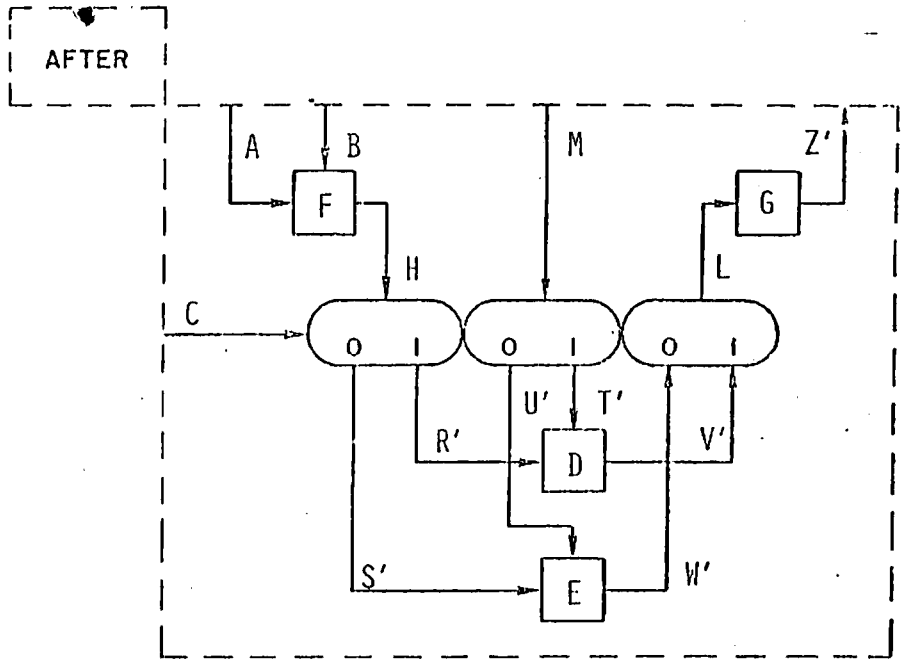
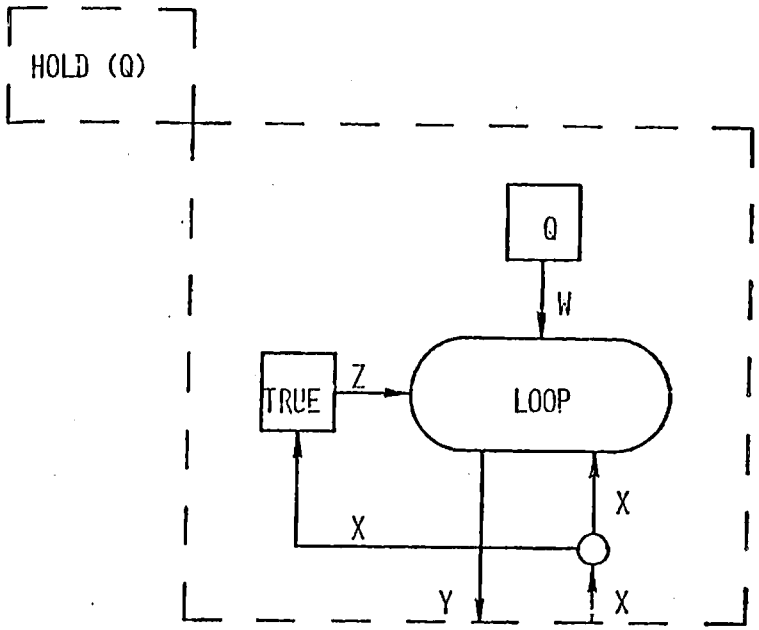




FIGURE 3




ABBREVIATION

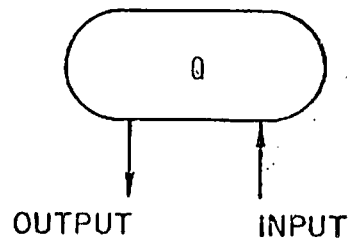



FIGURE 4

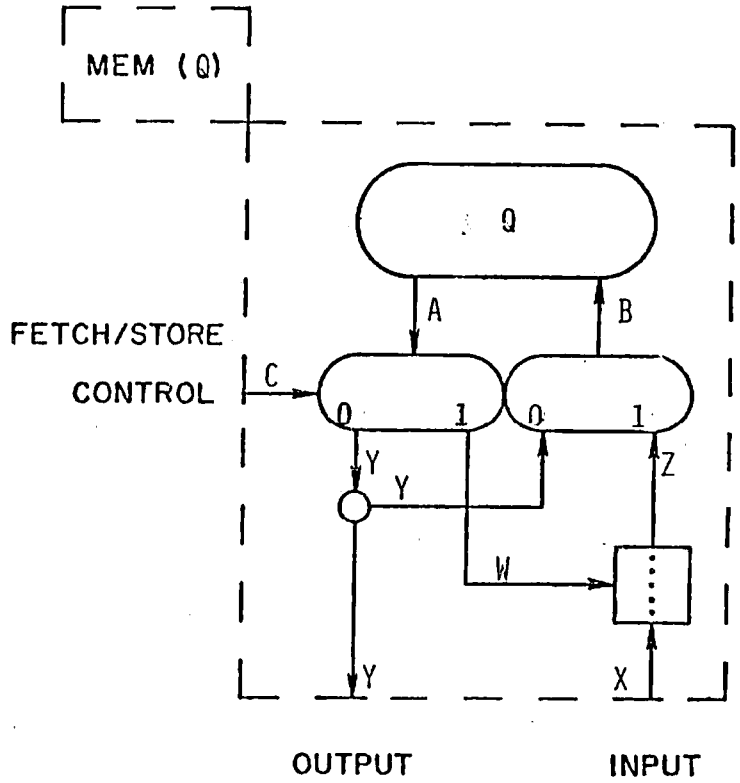



FIGURE 5