LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-121

THE EQUIVALENCE OF R. E. PROGRAMS AND DATA FLOW SCHEMES

Jeffrey Jaffe

January 1979

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-121

THE EQUIVALENCE OF R.E. PROGRAMS AND DATA FLOW SCHEMES

Jeffrey Jaffe

January 1979

MIT/LCS/TM-121

THE EQUIVALENCE OF R. E. PROGRAMS AND DATA FLOW SCHEMES

Jeffrey Jaffe

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

The equivalence of r.e. program schemes and data flow schemes

Jeffrey Jaffe, MIT *

Abstract. The expressive power of the data flow schemes of Dennis is evaluated. It is shown that data flow schemes have the power to express an arbitrary determinate functional. The proof involves a demonstration that "restricted data flow schemes" can simulate Turing Machines. This provides a new, simple basis for computability.

Keywords. data flow schemes, r.e. program schemes, effective functionals, Turing Machines, computability

1. Introduction

Early researchers investigating the relative "power" or "expressiveness" of different programming constructs quickly determined that a comparison of the set of partial recursive functions computed did not adequately capture the differences between the different programming styles. Any reasonable set of constructs computes all partial recursive functions, and thus all constructs are equivalent.

A different technique has evolved for comparing programming constructs. The notion of a scheme has been introduced [8,10,11], which enables one to discern differences between the classes of functionals computable by different constructs. Essentially, in a scheme there are no defined operations and thus

---

* This report was prepared with the support of a National Science Foundation graduate fellowship, and National Science Foundation grant no. MCS77-19754.

(for example) variables can not be used as counters. Rather, all function and predicate symbols are uninterpreted, and a "scheme" is a functional over the symbols. This approach turned out to be quite successful, as it provided a rigorous interpretation to intuitive ideas such as "recursion is more powerful than iteration" [12,15].

A hierarchy of program constructs has been developed [1,2]. It turns out that just as there is a notion of "all primitive recursive functions", there is an analogous thesis about a construct being equivalent to "all effective determinate functionals" or "all recursively enumerable (r.e.) program schemes" [14].

Data flow schemes were introduced by Dennis [3] to serve as a model of programming constructs to be used for highly parallel, data driven computer architectures. The power of this basic construct has never been fully explored. Early researchers in this area felt that it would be worthwhile to sacrifice the full power of this construct in order to enforce programming disciplines that are similar to those found in conventional languages. To this end, well formed data flow schemes were introduced [4]. They are a class of data flow schemes that satisfy certain structural requirements. These requirements force subprograms of any given program to behave like "if-then-while" statements. Indeed it has been shown [9] that the "expressive power" of well formed data flow schemes is equivalent to the expressive power of flowchart schemes.

Although data flow programs are often written using the well formed constraint, this constraint does not reflect the capabilities of the data flow computer architecture [5]. It is thus worthwhile to evaluate the expressive power of the entire class of data flow schemes which more closely approximate

the potential of such data driven, asynchronous architectures. The restrictions which give rise to well formed data flow schemes are a structure imposed from the outside, having little to do with machine architecture.

The main result of this paper is that the class of data flow schemes are equivalent to the class of effective determinate functionals. One direction is immediate using usual programming techniques in the language of r.e. program schemes. The proof of the other half is greatly simplified by two insights which are of interest in their own right.

The first is a programming technique using the language of data flow schemes. A general translation lemma is proved (using quite simple techniques) which characterizes a class of functions computable by data flow schemes but not computable by well formed data flow schemes. This lemma provides much of the machinery to prove later results.

The second insight is a theoretical result of interest. A very restrictive version of data flow schemes can simulate Turing Machines. This result implies various undecidability results about simple data flow schemes. The simulation of Turing Machines (TM's) follows immediately from the translation lemma.

In Section 2 r.e. program schemes, data flow schemes, and restricted data flow schemes are defined. Section 3 illustrates the programming techniques alluded to above. Using these techniques Section 4 discusses the simulation of TM's with restricted data flow schemes. Section 5 contains the proof that any r.e. program scheme can be simulated by a data flow scheme. Combining the programming techniques of Section 3 and the Turing Machine simulation of Section 4, this final simulation is not too hard to develop.

2. Syntax and semantics of schemes

2.1 R.e. program schemes

To define r.e. program schemes it is helpful to first define the components.

Any particular r.e. program scheme may have infinitely many *variable symbols* $x,y,u,v,\ldots$, finitely many *function symbols* $f_1,\ldots,f_r$ (from an infinite function symbol alphabet), and finitely many *predicate symbols* $p_1,\ldots,p_s$ (from an infinite predicate symbol alphabet). Associated with a function symbol f (or predicate symbol p) is a number $\text{arity}(f)\in\mathbb{N}$ which specifies the number of *arguments* needed by f. In the alphabet there is also a symbol *HALT*. Uninterpreted constants are thought of as 0-ary function symbols.

A *term* is:

(1) a variable x.

(2) a 0-ary function symbol f.

(3) the sequence $f(x_1,\ldots,x_n)$ where f is a function symbol of arity n, and $x_1,\ldots,x_n$ are variable symbols.

(4) the sequence $p(x_1,\ldots,x_n)$ where p is a predicate symbol of arity n, and $x_1,\ldots,x_n$ are variable symbols.

Terms of types (1), (2), and (3), are called *functional terms*. Terms of type (4) are called called *predicate terms*.

A *statement* is:

(1) a simple *assignment* $x\leftarrow t$ where t is a functional term and x is a variable.

(2) a predicate term.

(3) the symbol *HALT*.

An *r.e. program scheme*, P, is an infinite list of statements (indexed by

the integers) together with a recursive function $r : \mathbb{N} \times \{0,1\} \to \mathbb{N}$, a finite set of variable symbols $\{x_1,\ldots,x_n\}$ called *input variables* and a finite set of variable symbols $\{y_1,\ldots,y_k\}$ called *output variables*. Intuitively, the function r specifies the successor statement of a given statement.

2.2 Semantics of r.e. program schemes.

For brevity the presentation of this section will be a bit informal. For a more formal treatment for the semantics of schemes one may consult [6].

An *interpretation*, I, for a r.e. program scheme P consists of a domain D, an assignment of a total function from D^n to D to each n-ary function symbol, an assignment of a total predicate on D^n to each n-ary predicate symbol, and an assignment of an element of D to each input variable. A *program* is a pair (P,I).

The *computation* of a program (P,I) proceeds as follows. The first statement executed is statement 0 (the meaning of "executing a statement" should be understood - it involves updating variables or evaluating predicates). Assume that statement i has just been executed. If statement i was a $HALT$ statement, then the program is said to *terminate* and the *outputs* of the program are the current values of the output variables. If statement i was a predicate and the result of the predicate was $j \in \{0,1\}$, then the next statement to be executed is $r(i,j)$. Otherwise the next statement to be executed is $r(i,0)$.

2.3 Data Flow Schemes

A *data flow scheme* is a labelled (finite) directed graph (with self-loops and multiple arcs). The labels of the nodes of any particular data

flow scheme come from the following alphabets:

(1) An alphabet of function symbols $f_1, f_2, \ldots$

(2) An alphabet of predicate symbols $p_1, p_2, \ldots$

(3) The *gate* symbols "T", "F", "T F". (The gate labelled with "T F" is called a *Merge* gate.)

If a function symbol of arity n labels a node then there are n incoming arcs to the node. (Also, arity("T")=arity("F")=2 and arity("T F")=3).

Each arc in the graph is either a *boolean* arc or a *value* arc. (During execution each boolean arc has associated with it a word $w \in \{0,1\}^*$ and each value arc has associated with it a word $w \in D^*$ where D is a domain supplied by the interpretation). Each arc that leaves a node labelled with a predicate is a boolean arc. Each gate has a designated "control" boolean arc. (In the various figures, the control arc is labelled with a "c".) All arcs entering or leaving a node labelled with a function symbol are value arcs and all incoming arcs to a node labelled with a predicate symbol are value arcs. All of the noncontrol arcs that enter or leave a gate must be of the same type (but may be either boolean or value arcs). (Examples of various data flow schemes may be found in Figures 3.1-3.8. Their semantics are defined in Section 2.4.)

An *initialized data flow scheme* consists of a data flow scheme with an assignment of a word $w \in \{0,1\}^*$ to each boolean arc and an assignment of the empty word of D^* to each value arc.

A *restricted data flow scheme* is an initialized data flow scheme whose nodes are only labeled with "T", "F", and "T F".

Certain uninitialized arcs are designated *input arcs*. Certain arcs are designated as *output arcs*. It is sometimes convenient to allow two additional

labels for nodes. *INPUT* nodes have no incoming arcs, and *OUTPUT* nodes have no outgoing arcs.

2.4 Semantics of data flow schemes

An *interpretation* supplies a domain D, assigns functions to function symbols, and predicates to predicate symbols as in Section 2.2. Also, each input arc is assigned an input word from its value domain. A *program* is a scheme with an interpretation.

A node is said to be *enabled* if each of its incoming arcs has a non empty word associated with it. For *Merge* gates, the definition is slightly different: a *Merge* gate is enabled if the first symbol of the data word associated with its control input is T and its "T" arc is non empty or if its control input is F and its "F" arc is non empty. Also, *INPUT* and *OUTPUT* nodes are never enabled.

An enabled node *executes* by removing the first symbol of the word associated with each incoming arc, applying the function or predicate to the values represented by these symbols, and concatenating the result to the end of the words associated with each outgoing arc. For T gates, if the control arc is T, then the gate is the identity on the other incoming arc, and if the control arc is F then the first symbols are removed and nothing is placed on outgoing arcs. F nodes are the same as T nodes with the role of the control arc complemented. For *Merge* gates, if the control arc is T, the gate is the identity on the "T" arc and the "F" arc is unaffected. Similarly if the control arc is F.

At any step in execution there may be many enabled nodes. There is no notion of what "must" happen in one step, as the model does not completely

specify this. There is a notion of what may happen in one step. The work of [13] implies that this incomplete specification does not change the result of computations. At one step the data flow scheme executes between one and all of its currently enabled nodes. This updates some or all of the arcs, by removing old values from the data words associated with some of the arcs, concatenating result values to the associated data words of some of the arcs, doing both operations to some of the arcs, and leaving some of the arcs unchanged.

A data flow program *terminates* when no node is enabled. At that time the words associated with the output arcs are the *outputs* of the data flow program. (It is convenient to assume that formally, nodes labelled with constant functions (with no arguments) have one incoming arc, for otherwise a data flow program would never terminate.)

Note that as defined above, data flow schemes do not have a fixed number of inputs or outputs. When comparing data flow schemes to r.e. program schemes, the comparison considers only those data flow computations that have a fixed number of inputs and outputs in a fixed configuration on the input and output arcs. Thus each data flow scheme has associated with it integers which specify how many input symbols are to be supplied on each input arc and how many outputs are produced on each output arc. Alternatively, one may extend the definition of r.e. program schemes to permit a variable number of inputs and outputs. In that case, the simulation of r.e. program schemes by data flow schemes (given in Section 5) would have to be modified slightly, but not substantially.

3. A few programming techniques

There are two goals of this section. The first is to show that

restricted data flow schemes have enough power to define boolean functions (NOT and OR). This will be needed, as these functions play a role in subsequent results.

The second task is to prove the lemma alluded to in Section 1. This is a more substantial indication of the programming power of restricted data flow schemes. It is a useful tool in shortcircuiting the detailed programming needed to simulate TM's by data flow schemes.

3.1 Boolean operators

To define NOT and OR, consider the programs of Figures 3.1 and 3.2. In Figure 3.1, a True value on arc *A* chooses out the "*T*" arc of the *Merge* (i.e. the value False). A similar arrangement for *A* having False implies *B*=NOT(*A*). (The diagramming convention for arc *A* is an arc emanating from a node labelled with an *A*.

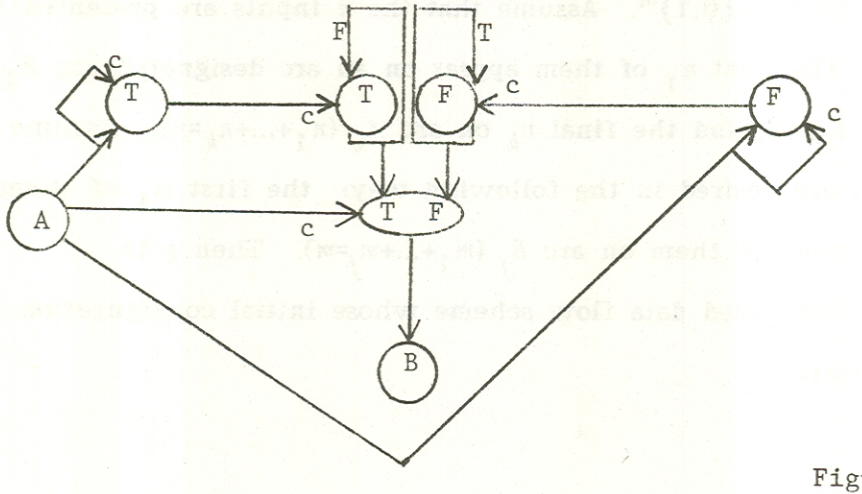


Figure 3.1

In Figure 3.2, when A is true, then A (i.e. True) is output, and when A is false then B is output. Thus $C=A$ OR B. Note that in both programs the initial configuration is restored after one usage. Thus values may be pieplined through these circuits, and the NOT circuit will always complement its input, and the OR circuit will always "OR" a pair of inputs.

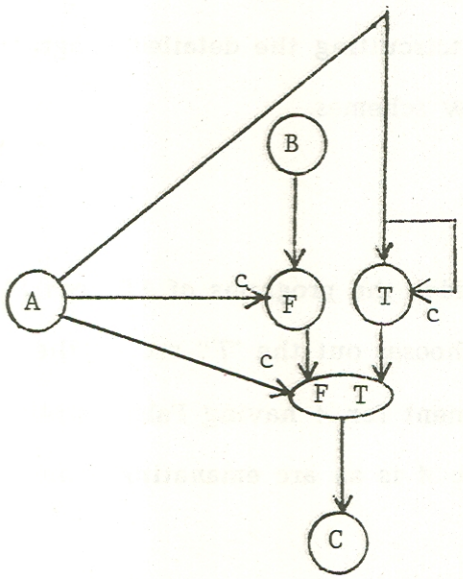


Figure 3.2

3.2. Finite translation Lemma.

Lemma. Let $g:\{0,1\}^n \to \{0,1\}^m$. Assume that the n inputs are presented in the following way: the first n_1 of them appear on an arc designated arc A_1, the next n_2 on arc A_2, ..., and the final n_k on arc A_k ($n_1+\ldots+n_k=n$). Assume that the outputs are desired in the following way: the first m_1 of them on arc B_1,..., the last m_j of them on arc B_j ($m_1+\ldots+m_j=m$). Then g is computable by a restricted data flow scheme whose initial configuration matches its final configuration.

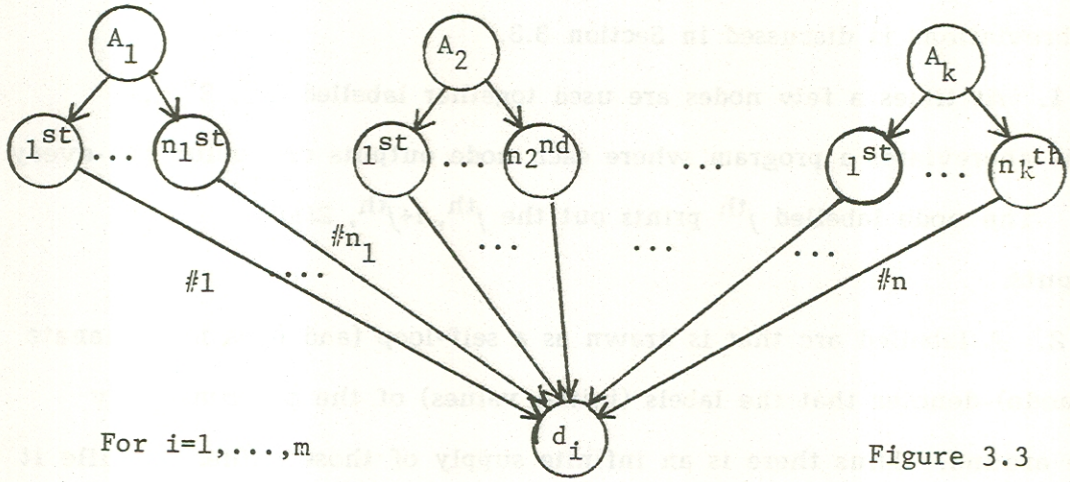
Proof The following explains certain abbreviations used in the data flow programs described in the proof (see Figures 3.3 and 3.4). The expansion of these abbreviations is discussed in Section 3.3.

1. At times a few nodes are used together labelled 1^{st}, 2^{nd},..., i^{th}. This abbreviates a program where each node outputs one value for every i inputs. The node labelled j^{th} prints out the j^{th}, $i+j^{th}$, $2i+j^{th}$ (etc.) inputs.

2. A labelled arc that is drawn as a self-loop (and does not emanate from a node) denotes that the labels (initial values) of the arc constantly circulate around. Thus there is an infinite supply of those values. While it may seem that a data flow program with such self-loops never terminates, the expansion of this abbreviation discussed in Section 3.3 does in fact terminate.

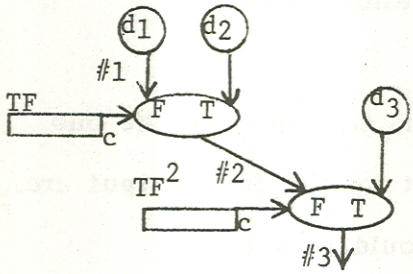
Now, consider Figure 3.3 which gives the first part of the desired data flow program. For each set of n inputs, arc #j obtains the j^{th} input. Thus the n inputs are "parallelized" onto n different arcs. The node labelled d_i is an abbreviation for an acyclic graph of boolean operators which produces the i^{th} digit of the m digit result. Note that all arcs are restored to their initial data words when the computation is completed.

For each of the j output arcs there is a program segment like the one in Figure 3.4. The figure denotes the program segment for the first output arc. The arc labelled d_i is 1 if the i^{th} of the m_1 results should be a 1. Arc #1 thus obtains the first of the m_1 results that will be output on arc B_1, arc #2 obtains the first two of the m_1 results, ..., and arc #m_1 obtains all m_1 results. Note that every arc has the same data word associated with it



For i=1,...,m

Figure 3.3



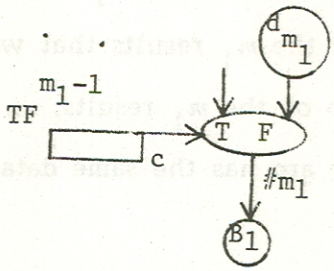


Figure 3.4

before and after each execution of the program. The self-loops are discussed in the next section. □

It is important that the initial configuration matches the final configuration. The program is thus reusable in the sense that if g must be applied repeatedly on a sequence of inputs, the same program may be used for each set of n inputs. Note also that the lemma is clearly false for well formed data flow schemes due to their well behaved characteristic [4].

3.3. Programs for the abbreviations

Figure 3.5 is the expanded version of the nodes labelled with i^{th}. If a node desires to choose the i^{th} of j values, it merely absorbs all but those of the form $nj+i$. This is accomplished by circulating around a control of $F^{i-1}TF^{j-i}$ to a T gate as in the figure.

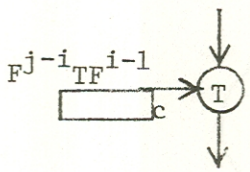


Figure 3.5

Figure 3.6 indicates the definition of an arc that is drawn as a self-loop. The arc actually represents two arcs leaving a T gate. One arc leads to the destination of the arc in the original program, and the other leads back to the input of the T gate. To control this T gate the program generates as many True values as needed. In particular, for the control inputs to the *Merges* of Figure 3.4, one would like i control values to the T gate for each value on d_i.

Figure 3.7 indicates how to accomplish this. The node labelled *xl*

expands one value on the input to l values on the output. Figure 3.8 is the $x13$ program and it is easy to generalize this to the construction of the program for xl for any l.

The self-loops of Figure 3.5 are easier to handle and are left to the reader.

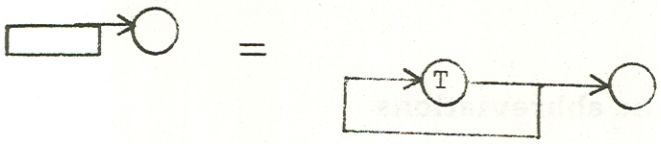


Figure 3.6

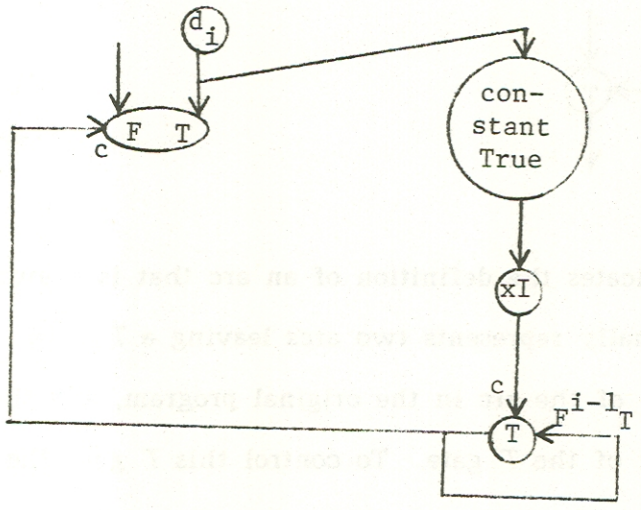


Figure 3.7

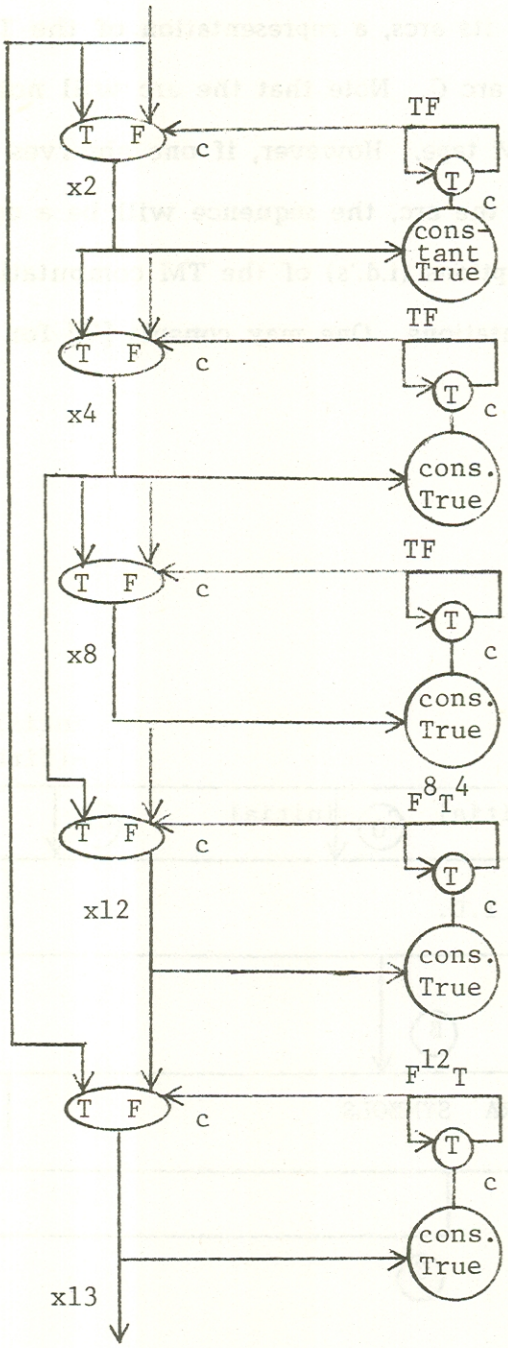

TF
T F
c
x2
T
c
cons-
tant
True
TF
T F
c
x4
T
c
cons.
True
TF
T F
c
x8
T
c
cons.
True
F8T4
T F
c
x12
T
c
cons.
True
F12T
T F
c
T
c
cons.
True
x13

Figure 3.8

4. Simulating Turing Machine computations with restricted data flow schemes.

The object of this section is to explain a simulation of one tape Turing Machines by restricted data flow schemes. The data flow scheme will simulate the TM by keeping on one of its arcs, a representation of the TM tape. In Figure 4.1, this data arc is labelled arc *C*. Note that the arc will not always have a representation of the TM tape. However, if one observes the sequence of symbols that pass through the arc, the sequence will be a coded form of successive instantaneous descriptions (i.d.'s) of the TM computation. We assume familiarity with TM computations. One may consult [7] for elaborations.

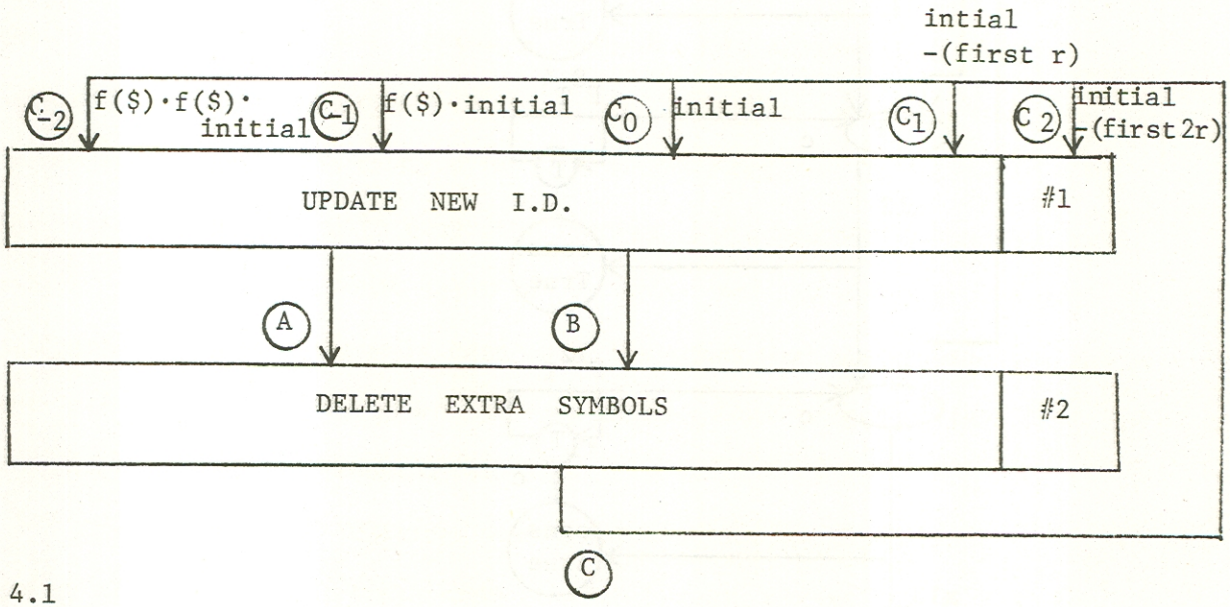


Figure 4.1

4.1. Representation

The following is an explanation of how the data flow simulator represents an i.d. of the Turing Machine computation. Let Σ be the tape alphabet of the TM and Q be the set of states of the TM ($\Sigma \cap Q = \varnothing$). Let $\$ \notin \Sigma \cup Q$ be a delimiter. Let $r = \lceil \log_2(|\Sigma|+|Q|+1) \rceil$. Then the data flow simulator represents an i.d. with respect to a fixed coding of the symbols of $\Sigma \cup Q \cup \{\$\}$ as r-tuples.

Let $f: \Sigma \cup Q \cup \{\$\} \rightarrow \{0,1\}^r$ be a 1-1 map. Assume that the active portion of the TM's tape is $w_1 \cdots w_k$ with $w_i \in \Sigma$ (the active portion of the tape consists of those squares already traversed), and that the machine is in state q reading w_j. Then the f-code of the i.d. is given by $f(\text{i.d.}) = f(w_1) \cdot f(w_2) \cdots f(w_{j-1}) \cdot f(q) \cdot f(w_j) \cdots f(w_k) \cdot f(\$)$. Note that the r symbols in the f-code that appear before the coded version of the symbol currently being scanned, is the coded form of the state.

The f-code of the initial i.d. is f(start state) followed by the coded versions of the input to the TM, followed by $f(\$)$. If the entire tape is initially blank then the f-code of the initial i.d. is $f(\text{start state}) f(\not b) f(\$)$ where $\not b$ is the blank symbol. The f-code of the computation of a TM M on an input, is the concatenation of the f-codes of the individual i.d.'s (starting with the inital i.d. and ending at the halt i.d.).

Assume without loss of generality that in one step a TM will either move left, move right, print $\sigma \in \Sigma$, read and branch to a state based on what was read, or halt.

If the TM is reading the leftmost active square and moves left, or the rightmost active square and moves right, then the simulator must know to expand its representation of the TM's tape. (If the TM head moves right and ends up scanning a square not yet activated the data flow simulator does not activate

the scanned square until that square is written on, or until the read head moves to the right of it.) Otherwise, to update an i.d., only a few symbols must be changed, with none added.

4.2. Description of simulation

Definition. A restricted data flow scheme D with input arc C, *simulates* (with respect to f) a TM M if for any input to M, the sequence of symbols passing through C is the f-code of the computation of M on that input (when the input to D is the assignment of the f-code of the initial i.d. of M to C).

For convenience, assume that any TM has infinitely many i.d.'s. The successor i.d. of a halt i.d. is the halt i.d.

Theorem 1. For any TM M, there is a restricted data flow scheme D that simulates M.

Proof. Consider Figure 4.1. Given a particular i.d., assume that the simulator has the f-code of this i.d. on arc C. It suffices to show that the next sequence of symbols to be added to C is the f-code of the successor i.d. of the current i.d. Then, to prove the theorem, the program represented by Figure 4.1 will be used, with input arc C. Note that the nodes of the data flow graph will not necessarily execute at the times that are assigned to them in this discussion, but due to determinacy of data flow programs, this does not effect the result of the computation [13].

Arc C feeds into different arcs denoted $C_{-2}, C_{-1}, C_0, C_1, C_2$. Initially, all five arcs have the same associated word, except that C_{-2} has an extra $2r$ symbols ($f(\$)\cdot f(\$)$), C_{-1} has an extra r symbols ($f(\$)$), C_1 is

missing the first r symbols of the i.d. and C_2 is missing the first $2r$ symbols of the i.d. (It is left as an exercise how to remove the first r symbols to get C_1. It is not hard to do if one uses a T gate with an initial control of r 0's. Care must be taken to insure that no further symbols are deleted. To get the $f(\$)$ values initially, just use an initialization of arc C_{-1} to $f(\$)$ and arc C_{-2} to $f(\$)\cdot f(\$)$.) Box #1 of Figure 4.1 updates the current i.d. by processing r symbols from each arc at once. If the first r symbols on arc C_0 are the representation of the contents of a TM square, and the square is "far" from the square currently being scanned, then these r symbols are copied to the next i.d. Similarly, if these r symbols are $f(\$)$ and the TM head is far from the extreme left or extreme right of the active portion of the tape, then $f(\$)$ is copied.

One might wonder how it is possible for the program to know whether the head scans a nearby square. This information is contained on arcs $C_{-2},\ldots,C_2$. Recall that arc C_i is initialized so that at all times its first block of r symbols is the block that will appear i blocks later in the i.d. then the current block of C_0. (If i is negative, then it is the block that appeared $-i$ blocks earlier.) As will presently be discussed, the changes in the i.d. may be determined from these five blocks of r symbols.

To finish the discussion of the updating, the following is done if the read head is near the TM symbol coded by the r bits on arc C_0. First consider the case that the i.d. need not be expanded (due to the exploration of new squares on the extreme right or left of the tape). If the TM symbol coded by the block of r symbols on C_0 is the predecessor of the square being scanned, and the TM is in a left move state, then the r symbols output by box #1 are the r symbols of the representation of the next TM state. If the r symbols are the

code of a left move state, then the output is the first r symbols on arc C_{-1}. Similarly, it is easy to see how to update the i.d. for any type of TM state. For example, if arc C_0 contains the code of a read and branch state, then the output is the code of the new state based on the first block of r symbols on C_1 (i.e., the TM symbol being scanned). If arc C_0 is the code of a print state then the output is the code of the next state. If arc C_{-1} is the code of print σ, then the output is $f(\sigma)$. A right move is similar to a left move except that the location of the state symbol is interchanged with the *next* symbol of the f-code of the i.d.

The final cases to consider are the cases of a left move onto a new square, a right move onto a new square, and a print onto a new square. A left move onto a new square is recognized when C_{-2} has $f(\$)$ and C_{-1} has the code of a left move state. In that case, the output is $f(\not\flat)\cdot g$ where g is the input on arc C_0 ($\not\flat$ is the new leftmost symbol). If C_0 has $f(\$)$ and C_1 has the code of a left move state, then the output is $f(\$)$, and if C_0 has a left move state and C_{-1} has $f(\$)$ the output is the code of the next state. In the right move case, the r symbols that coded the rightmost square are updated to $2r$ symbols. In the print σ case, the block $f(\$)$ is updated to $f(\sigma)\cdot f(\$)$. Note that in these cases $2r$ symbols are output, whereas in other cases only r symbols need to be output.

For conformity to the hypothesis of the finite translation lemma, it is convenient to assume that box #1 always gives the same number of results on each arc for each set of 5 input blocks. Thus, box #1 always prints out $2r$ symbols on arc A. Arc B consists of 1^{2r} when all $2r$ symbols on arc A are desired, and $0^r\cdot 1^r$ if only the last r are desired. In that case, the first r are arbitrary (included on arc A for convenience), and the last r are the

desired r symbols. The irrelevant symbols are deleted in box #2.

As far as the actual program is concerned, there is not much to add. Careful inspection of the specifications of box #1 indicates that it fits the hypothesis of the finite translation lemma, and thus a program exists for it. The program for box #2 is trivial, and is given in Figure 4.2. □

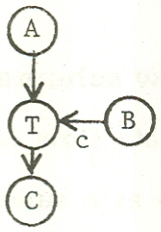


Figure 4.2

4.3. Halting

The above discussion has not considered the results of computation in the case that the "TM" has a notion of output. This problem has been ignored since the main motivation for our simulation has been to help prove that data flow schemes are as powerful as arbitrary r.e. program schemes. For that purpose, showing that a data flow scheme can simulate the control structure of a TM is sufficient.

A brief description of a possible convention will be discussed. Whenever C_0 has f(halt state), box #1 outputs f(halt state). In addition, if C_{-1} has f(halt state) then the r symbols that appear on arc C_0 are deleted (i.e. arc B consists of 0^{2r}). Ultimately, one of the C_i becomes void, causing the data flow program to terminate.

Now assume that box #1 also contains two additional outgoing arcs (arcs D and E) where D ordinarily produces 0^r and instructs a T gate to delete r symbols output on arc E. When the halt i.d. is reached, at each step that the

data flow simulator deletes r symbols, arc D becomes 1^r, permitting the r deleted symbols (that are now sent to arc E) to be preserved.

Now, assume that the desired output convention is that the sequence of symbols until the first b, is the output. The symbols that have just been deleted from the i.d. representation are the input to another data flow program which outputs its input until it sees $f(b)$. From that time on, no output is produced. Such a program may be constructed and is left to the reader.

5. Simulation of an arbitrary scheme with a data flow scheme

To simulate a particular r.e. program scheme, the data flow scheme will use a particular Turing Machine (as simulated in Section 4) as a subroutine. The TM that is chosen is one that is capable of generating the actions to be performed by the program scheme given the current statement number of the program scheme and (where relevant) the results of predicates.

The TM interfaces with a data flow program called "the scheme simulator". This program contains the current values of the variables of the r.e. program scheme (that have already been defined) on a "value list" (one of its arcs), and also has nodes labeled with the uninterpreted function and predicate symbols. The TM instructs the scheme simulator when to use these nodes and how to update the value list. In order to do this, the TM also maintains an "association table" which keeps track of which variables occupy which positions in the scheme simulator's value list at any particular time.

5.1. Operation of the Turing Machine.

Initially, the Turing Machine has a blank tape. The first task of the TM is to initialize the association table. If the n input variables to the

scheme are $x_1,...,x_n$, the table contains information that "for i=1,...,n, the i^{th} value on the scheme's value list is x_i".

Assume that the TM has the number of the last statement executed (initially 1) written on its tape (referred to as i). Assume also that if statement i was a predicate, that the result of the predicate is currently on the TM tape. The TM then computes the number of the successor of statement i in the program scheme using the recursive function that generates the next statement number. (Note that the association table must be left intact in the process.) The Turing Machine then computes the type of operation required by $s(i)$, the successor of instruction i. Assume that all variables required as input to the operation have already been defined (i.e. appear in the table), otherwise, the TM loops.

Let the operation required by statement $s(i)$ be $Z(y_1,...,y_k)$. The TM now instructs the scheme simulator where to get $y_1,...,y_k$ from, which function or predicate is referred to by Z, and where to place the result. First the TM sends a message that tells the scheme simulator what to do with the top of the value list. Specifically, if it equals y_i for some i, the message is "Let the top value be used as the i^{th} input to the operation Z and circulate the top value to the end of the value list". If it equals none of the y_i then the message is "Circulate the top value to the end of the list". At this time the TM updates its association table so it knows which value on the list refers to which variable.

The way that messages are sent is as follows. There are finitely many possible messages, and each is coded as a different symbol of the TM's tape alphabet. These symbols are reserved symbols used only for this purpose and are only printed by the TM, when messages are to be sent. The step following

the printing of a reserved symbol, is always a print σ for some nonreserved symbol σ. Thus the same "message" does not exist in two successive i.d.'s. The motivation for this will become clear when the operation of the scheme simulator is discussed.

The TM continues sending messages to the scheme simulator, until all the inputs to Z have been defined.

After Z has received all inputs, subsequent messages take on one of five forms.

(1) If Z is a function symbol, and the action required by statement $s(i)$ is $x_n \leftarrow Z(y_1,\ldots,y_k)$ where x_n has not yet been defined, then the message sent is "Add the result of Z to the list of values".

(2) If the statement was a function application $x_n \leftarrow Z(y_1,\ldots,y_k)$ and x_n has already been defined, then if x_n is the top value on the list, the message says "Replace the top value with the result of Z".

(3) In the case of a function application, assigned to an already defined variable that is not the top of the list, the message is "Circulate the top of the value list to the end of the list". In this case, the TM then proceeds to determine if the new top of the value list is x_n.

In all of the above cases, the TM updates its association table.

(4) If Z was a predicate, then the next message is "Obtain the result of the predicate Z". Then, a "return message" is sent that says "Return the result of a predicate to the TM". The return is accomplished as follows. The "return message" sent by the TM is some symbol $\sigma \in \Sigma$. Let $\sigma' \in \Sigma$ be a reserved symbol of Σ whose code differes from that of σ in exactly one place. (Assume that σ has a 0 in that place, and σ' has a 1 in that place.) Then the scheme simulator leaves σ unaffected if the predicate was false, and changes the code

of the message from σ to σ' if the predicate was true.

(5) If Z is a $HALT$ instruction then getting the values for $y_1,\ldots,y_k$ (in the above discussion) is all that is necessary, and the scheme simulator outputs those values.

After operation $s(i)$ is completed, the TM continues to $s(s(i)))$. If $s(i)$ is a $HALT$ operation, then after $s(i)$ is completed, the TM halts. Using techniques similar to those of Section 4.3, it is easy to see how to make the data flow version of the TM halt.

5.2. Operation of the scheme simulator

The outline for the program of the scheme simulator is given in Figure 5.1. The scheme simulator is given as input the n initial values of the r.e. program scheme being simulated (on arc G in Figure 5.1). The scheme simulator uses the TM described in Section 5.1 as follows. Consider Figure 4.1. Arc C (for the TM) is interrupted and passed as input to the scheme simulator. The scheme simulator returns an output to the TM which is identical to the input unless the result of a predicate needed to be passed. In this way, the scheme simulator sees every symbol of every i.d. exactly once.

If the scheme simulator sees a reserved symbol, then the simulator knows that action must be taken at this i.d. of the TM. Otherwise the simulator does nothing.

The possible actions are:

(1) Use the top value of the value list as input to a certain function or predicate.

(2) Update the top value without using it.

(3) Change the top value according to the result of a function.

(4) Add the result of a function to the value list without deleting the top value.

(5) Evaluate the result of some predicate.

(6) Return the result of the last predicate evaluated.

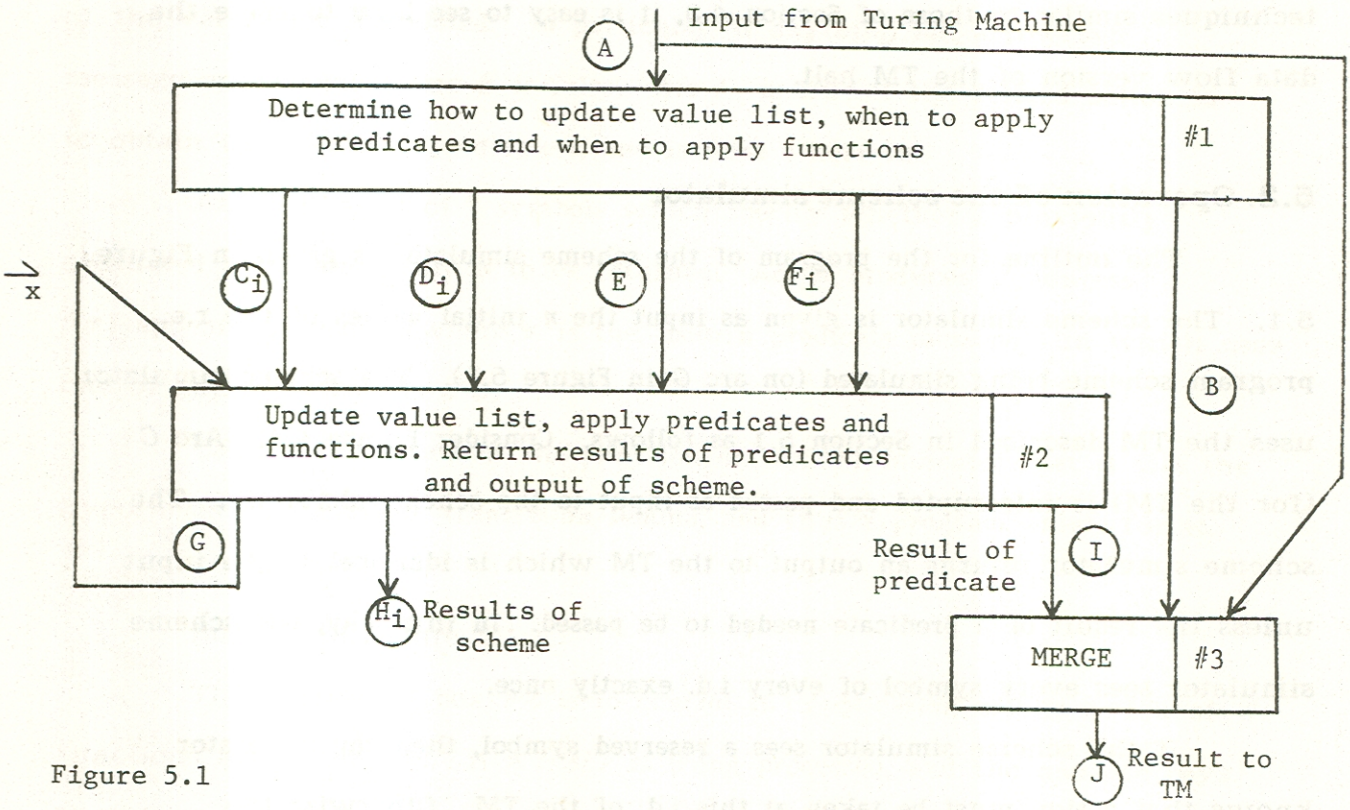


Figure 5.1

Once it is shown how to write the program for the scheme simulator it is easy to prove the main theorem:

Definition Let P be an r.e. program scheme, and D be a data flow scheme with input arc G. Then P and D are said to be *equivalent* if for all interpretations of all the function and predicate symbols of P and D and all inputs to P, P halts iff D halts, and the output of P equals the output of D (when the input to D is the ordered set of inputs to P placed on arc G).

Theorem 2. Let P be a r.e. program scheme. Then there is a data flow scheme D that is equivalent to P.

Proof. Use the TM described in Section 5.1 appealing to the construction of Section 4. It suffices to show how to implement the above description for the scheme simulator. The following is a detailed description of the program for the scheme simulator.

Box #1 of Figure 5.1:

Arc A of the flowchart of Figure 5.1, is the input from the data flow program which simulates the TM described in Section 5.1. The input is used in the following way. The scheme simulator searches through the i.d.'s for reserved message symbols. The function of box #1 is to determine which operations must be performed by the scheme simulator.

Inputs to box #1 are processed r symbols at a time. When arc A has $f(\sigma)$ where σ is a reserved message symbol, box #1 decides which action needs to be performed for this i.d. The possible actions are as follows. If the top

value of the value list is to be used at the i^{th} position that values are used, then arc C_i is true. Otherwise arc C_i is false. (Each possible arc that could use a value from the value list is assigned a position). If the result of function f_i is to be added to the value list then arc D_i is true, otherwise arc D_i is false. When one of the D_i is true, then arc E is true if the result of the function is to replace the top of the value list and is false if the result is added to the value list. In general, arc E is true if the top of the value list is updated. In particular, if anything but a reserved message appears on A, arc E is false. Arc F_i is true if the desired action is to obtain the result of predicate i, and is false otherwise.

Arc B is a set of r symbols which specify whether a previously evaluated predicate result is to be returned. Arc B produces 1^r, unless $f(\sigma)$ appears on arc A (where σ is the "return predicate message") in which case one of the r symbols is a 0.

Careful inspection of the function of box #1 indicates that it fits the hypothesis of the finite translation lemma, and thus a program exists for it.

Box #2 of Figure 5.1:

The purpose of box #2 is to control the application of the action of functions and predicates. If a *HALT* is being processed, and the message says to output the top value of the value list on the i^{th} output arc, then box #2 outputs the top value of the value list on arc H_i. If the top value is to be updated, then arc G is modified according to whether a new value is added to the list, the top value is changed, or the top value is recirculated. If the top value is to be sent to an action Z as an input, then that action is controlled here. If the result of some predicate is to be sent to the TM, it

is sent on arc *I*.

The program for box #2 is given in Figure 5.2. For each possible function or predicate there is a subprogram described in Figure 5.2a. For the *HALT* action there is a subprogram described in Figure 5.2b. The subprogram for each possible function or predicate is as follows. Assume that arc *G* contains the value list and assume that *Z* is a function (or predicate). For each input of *Z* (if any) that needs the top value, the *C* values corresponding to that position will be "True", passing through the top value. The gating accomplished with arc *E* merely insures that the top value will not be lost if it is not supposed to be looked at during this step. The arc labelled out(*Z*) obtains the result of the operation after all inputs to *Z* have arrived.

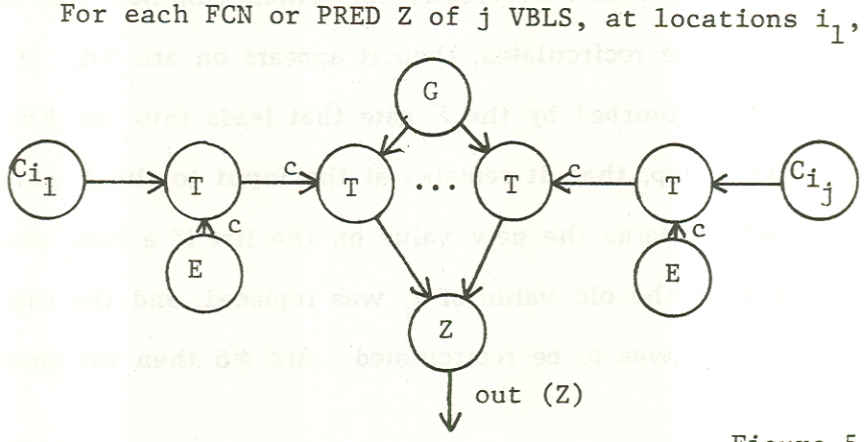


Figure 5.2a

Figure 5.2b is similar to 5.2a. If a *HALT* is the desired operation, then the meaning of "needing" the top value as the i^{th} input to *HALT*, is that arc H_i should get the top value (i.e., the top value is the i^{th} output value).

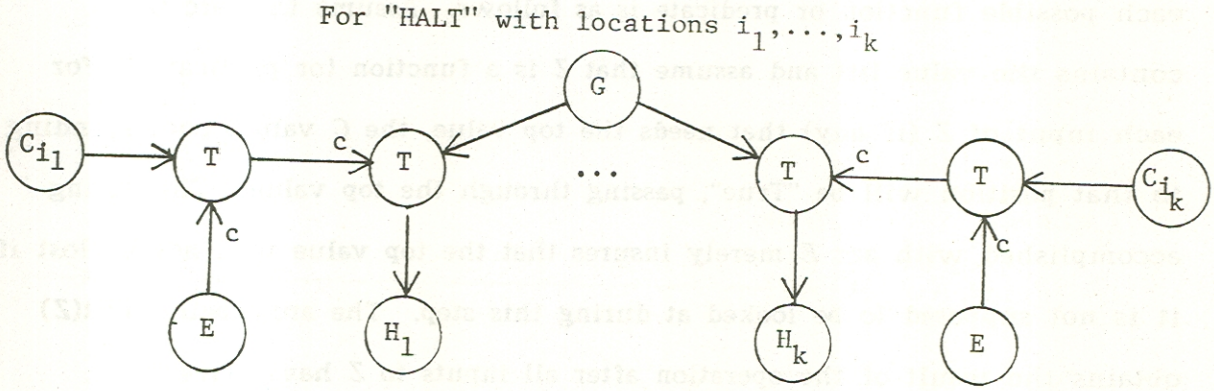
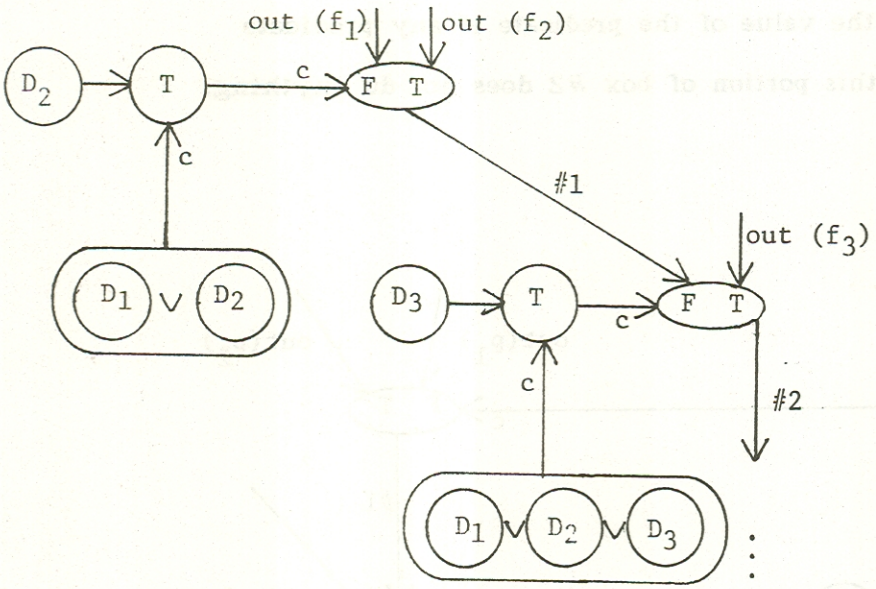


Figure 5.2b

Figure 5.2c describes the updating of the value list. If a function has been applied, then arc #1 is the value of the function if the operation was an f_1 or f_2. Similarly, arc #2 is the result if the operation was f_1, f_2, or f_3, and arc #3 is the result irrespective of which function was applied. If the top value is to be recirculated, then it appears on arc #4. If it is to be replaced then it is absorbed by the *F* gate that leads into arc #4. If it is to be ignored at this step, then it remains at the input to the *F* gate. After an operation arc #5 contains the new value on the list if a new one was added, the new value of x_i if the old value of x_i was replaced, and the old top value, if the top value was to be recirculated. Arc #5 then becomes the end of the value list.


out (f_1)
out (f_2)
D_2
T
c
F T
c
#1
out (f_3)
D_1 ∨ D_2
D_3
T
c
F T
c
#2
D_1 ∨ D_2 ∨ D_3

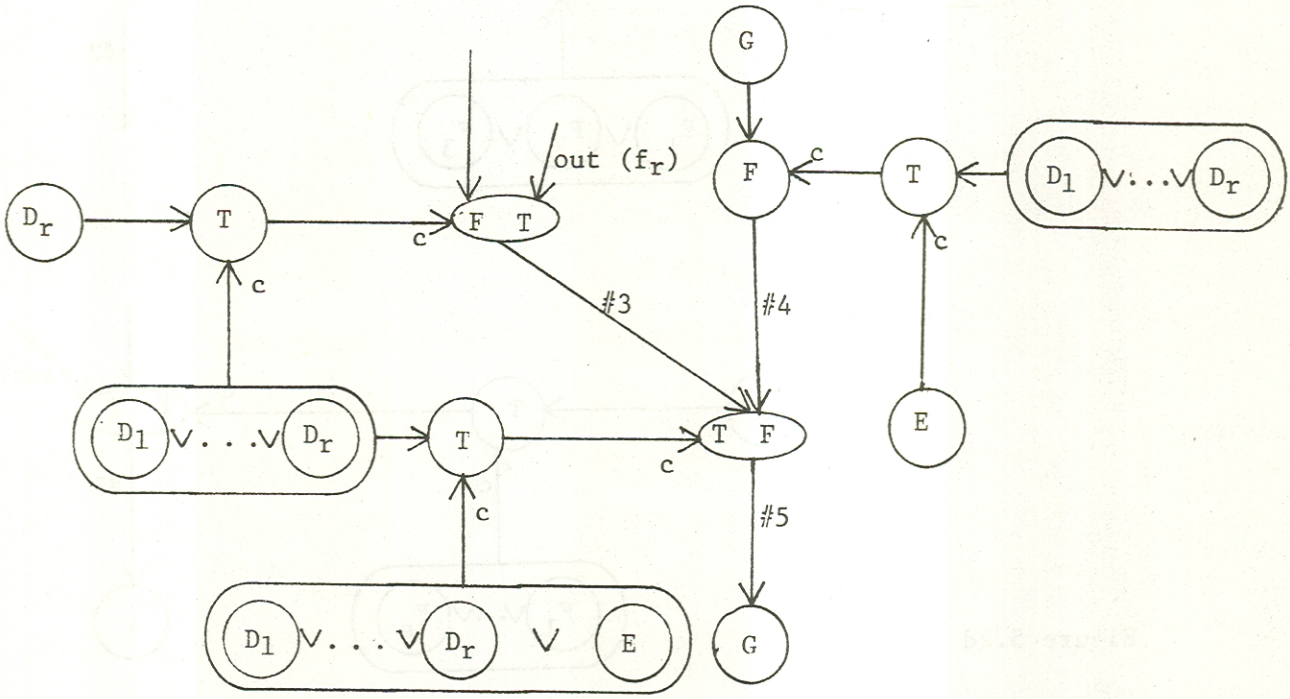

G
out (f_r)
F
c
T
D_1 ∨ . . . ∨ D_r
D_r
T
c
F T
c
c
#3
#4
D_1 ∨ . . . ∨ D_r
T
T F
E
c
c
#5
D_1 ∨ . . . ∨ D_r ∨ E
G

Figure 5.2c

Figure 5.2d is similar to 5.2c in that it merges the predicate results. Arc #1 is the value of the predicate if it was the first or second predicate that applied, arc #2 is the value of the predicate if any of the first three predicates applied, and arc #3 is the value of the predicate if any predicate applied. If no predicate applied, this portion of box #2 does not do anything.

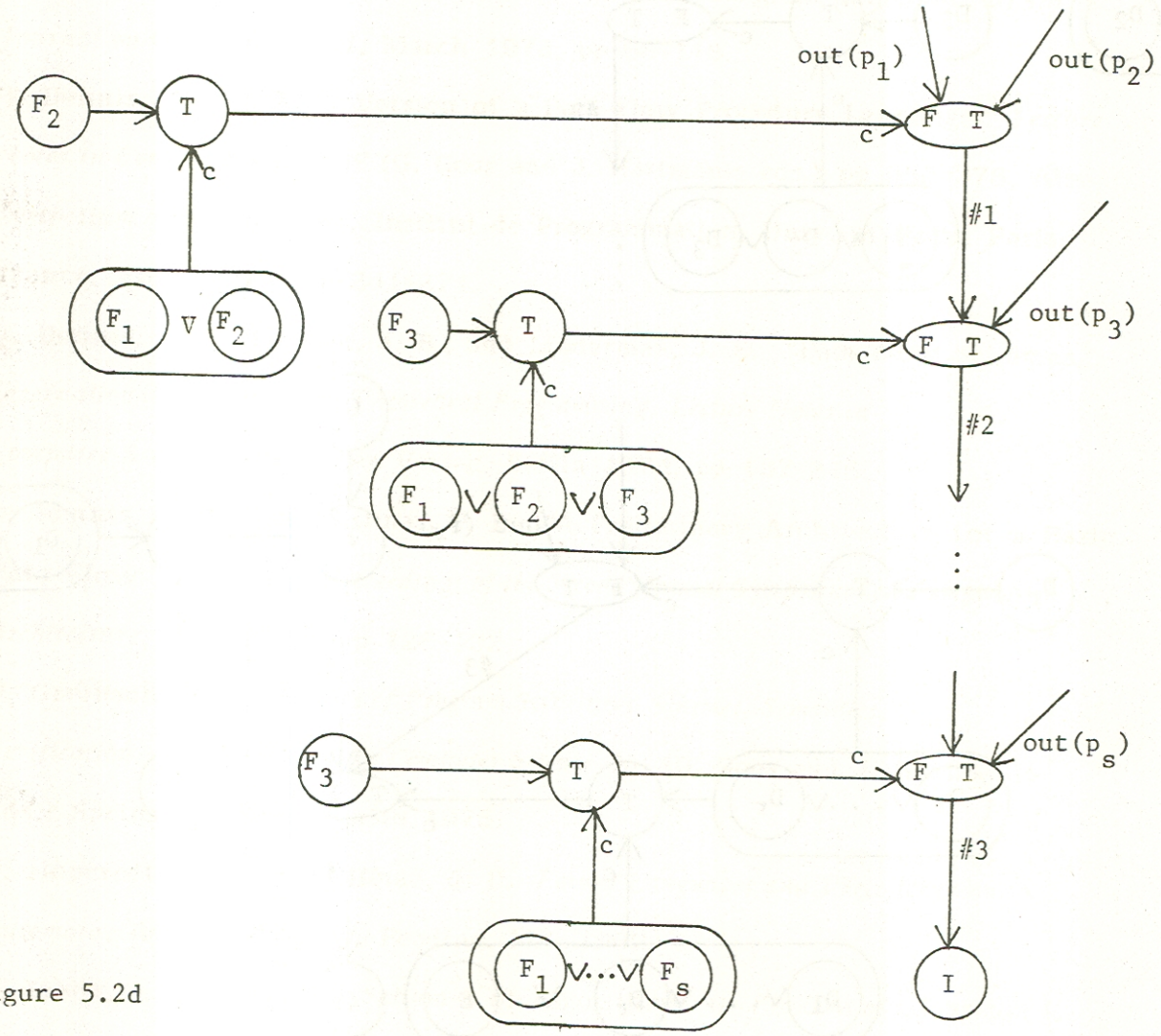


Figure 5.2d

Box #3 of Figure 5.1:

The purpose of box #3 is to merge the old TM i.d. with the result of the predicate if applicable. If arc B has 1^r then the block from arc A is output by the scheme simulator. If arc B has one 0, then arc I is substituted for one of the result bits. The program for box #3 is given in Figure 5.3.

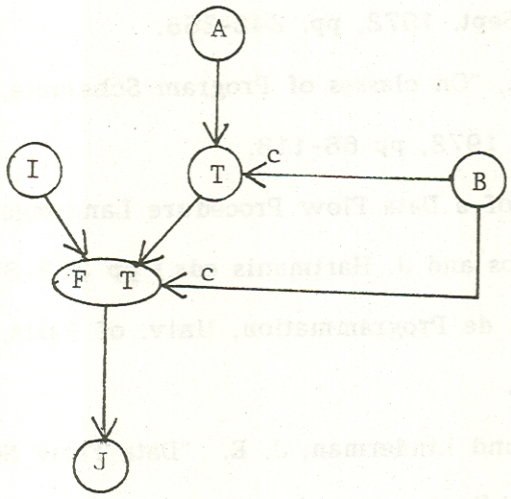


Figure 5.3

If the set of arcs $\{H_i\}$ are the output arcs of this data flow program, then the above simulation successfully simulates P. □

6. Conclusion

The power of two versions of data flow schemes have now been analyzed. There is a wide gap between well formed data flow schemes which are almost a direct translation of "if-then-while" programs [9], and data flow schemes which fully express the architectural constructs of data driven architectures [5]. It would be interesting to define natural restrictions on data flow schemes which make a subclass of data flow schemes equivalent to other models in the scheme hierarchies [1,2].

Acknowledgements

The author wishes to express his gratitude to Albert Meyer for several helpful suggestions which greatly simplified the constructions of this paper.

References.

1. Brown, S., Gries, D., Szymanski, T., "Program Schemes with Pushdown Stores," *SIAM Journal on Computing, 1*, 3, Sept. 1972, pp. 242-268.

2. Constable, R. L., and Gries, D., "On classes of Program Schemata, *SIAM Journal on Computing, 1*, 1, March 1972, pp 66-118.

3. Dennis, J. B. "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science 19* (G. Goos and J. Hartmanis eds.) pp 362-376, Also *Symposium on Programming*, Institut de Programmation, Univ. of Paris, Paris, France, April 1974, pp 241-271.

4. Dennis, J. B., Fosseen, J. B., and Linderman, J. E. "Data Flow Schemas," *International Symposium on Theoretical Programming, Lecture Notes in Computer Science 5*, Springer Verlag, Berlin 1974, pp 187-216.

5. Dennis, J. B., and Misunas, D. P. "A Preliminary Architecture for a Basic Data-Flow Processor," *Proceedings of the Second Annual Symposium on Computer Architecture*, Jan. 1975, pp 126-132.

6. Greibach, S. A. *Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science 36*, (G. Goos and J. Hartmanis Eds.) Springer Verlag, Berlin 1975.

7. Hopcroft, J. E. and Ullman, J. D. *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

8. Ianov, I. "The Logical Schemes of Algorithms" in *Problems of Cybernetics 1*, pp. 82-140, Perganon, NY 1960.

9. Leung, C. K. "Formal Properties of Well-Formed Data Flow Schemas," MIT, LCS, TM 66, Cambridge, MA., June 1972.

10. McCarthy, J. "Towards a Mathematical Science of Computation" pp 21-28, *Proceedings of IFIP Congress*, Munich 1962.

11. Paterson, M. "Equivalence Problems in a Model of Computation," Ph.D. Thesis, Univ. of Cambridge.

12. Paterson, M. and Hewitt, C. "Comparative Schematology, "*Record of the Project MAC Conference on Systems and Parallel Computations*, ACM, New York, 1970, pp 119-128.

13. Patil, S. S. "Closure Properties of interconnections of determinate systems," *Record of the Project MAC Conference on Systems and Parallel Computations*, ACM, New York, 1970, pp 107-116.

14. Strong, H. R. "High level languages of Maximum Power," *Proceedings of Twelfth IEEE Conference on Switching and Automata Theory*, 1971, pp 1-4.

15. Strong, H. R. "Translating Recursion Equations into Flowcharts," *JCSS*, 5, (1971), pp 254-285.