SOME RELATIONSHIPS BETWEEN ASYNCHRONOUS INTERPRETERS OF A DATAFLOW LANGUAGE*

by

Arvind
Kim P. Gostelow

Technical Report #88A

REVISION

*This work was supported by NSF Grant MCS76-12460:
The UCI Dataflow Architecture Project.

Z
699
C3
no. 88A

To be published in the:

Proceedings of the Working Conference in the Formal Description of of Programming Concepts held 1-4 August 1977, St. Andrews, Canada, under the auspices of IFIP Technical Committee 2 (Programming Languages).

SOME RELATIONSHIPS BETWEEN ASYNCHRONOUS INTERPRETERS OF A DATAFLOW LANGUAGE*

Arvind and Kim P. Gostelow
Department of Information and Computer Science
University of California, Irvine
Irvine, California

The theory of fixpoint semantics is applied to discover relationships between different interpreters of a dataflow programming language. A very concise model of the relative asynchrony of two interpreters is given and it is shown that one interpreter (described in the paper) is the most asynchronous interpreter possible for the given language. This same interpreter may also produce more results than other less asynchronous interpreters when executing the same program. Conditions are also given, which if followed by an actual machine architecture, will guarantee that the machine will always compute the least fixpoint of the program regardless of the order in which computation steps are carried out or the availability of processing resources.

1. INTRODUCTION

This paper applies the theory of fixpoint semantics to the design of computer architectures and operating systems. Our particular interest is in asynchrony (or concurrency, or parallelism) of execution, and our starting point is the adoption of a dataflow language [2,5] as the base machine language to be directly executed by a new computer system [1]. Dataflow languages are single-assignment languages [3,4,8,10,13] that offer many advantages over more conventional languages in areas such as modularity, inherent asynchronous execution, and more easily describable semantics. We have selected Dennis' Data Flow (DDF) language [5] as the base language for our architecture because of its relatively advanced state of development [6,13] over other dataflow languages.

We begin by specifying a formal semantic interpretation of DDF, where the semantics is best considered a model of a machine which would execute DDF. (The model is, in fact, a slight generalization of the machine proposed by Dennis [7] for executing DDF.) We call this first interpreter the Queued Interpreter (QI) since it treats the input and output lines of an operator as FIFO queues. We then give a second semantic interpretation of DDF corresponding to a second machine [1] called the Unfolding or Unraveling Interpreter (UI) which does not explicity use FIFO queues. Using the relation "less defined than or equal to" on the history of lines (variables) as the basis of our measure of asynchrony, we show that for any program in DDF the UI machine gives results that are more defined than or equal to QI, and

*This work was supported by NSF Grant MCS76-12460.

that UI is capable of more asynchrony (concurrency of execution) than QI. Furthermore, we extend the above comparison of UI and QI and show that no interpreter operating under the basic dataflow rules can produce greater concurrency of execution than UI, unless that interpreter is allowed to "guess" the result of computations. Lastly, we show that if a machine operates under three particularly simple conditions, then we can guarantee that regardless of how slowly it progresses (due, say, to a reduction in available resources), or the order in which it carries out its allowed work, the least fixpoint will be the natural result of its computation.

2. THE DATAFLOW LANGUAGE

Figure 1 shows the operation of the dataflow operators: function and

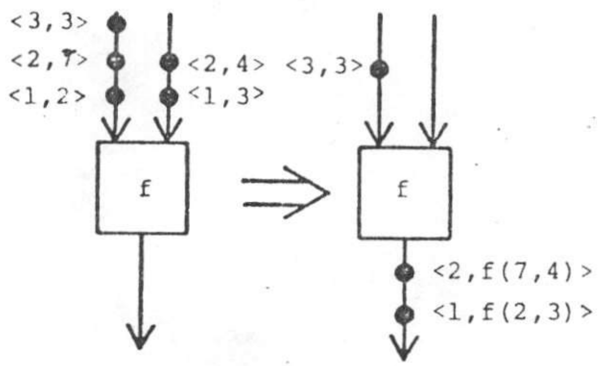


Figure 1a

The result of all firings of a function or predicate operator f under QI

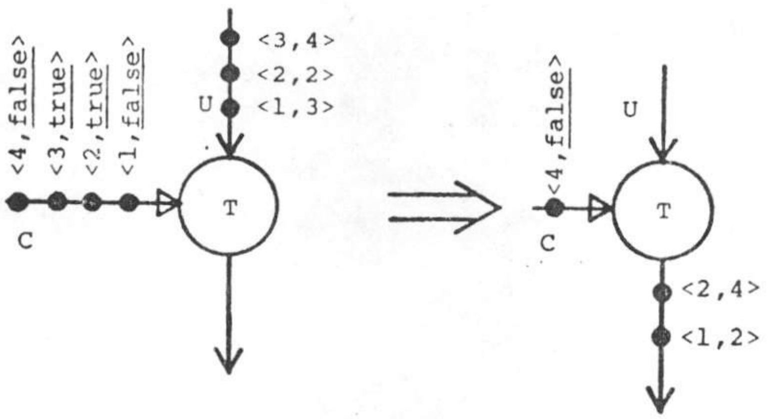


Figure 1b

The results of a gate-if- true operator

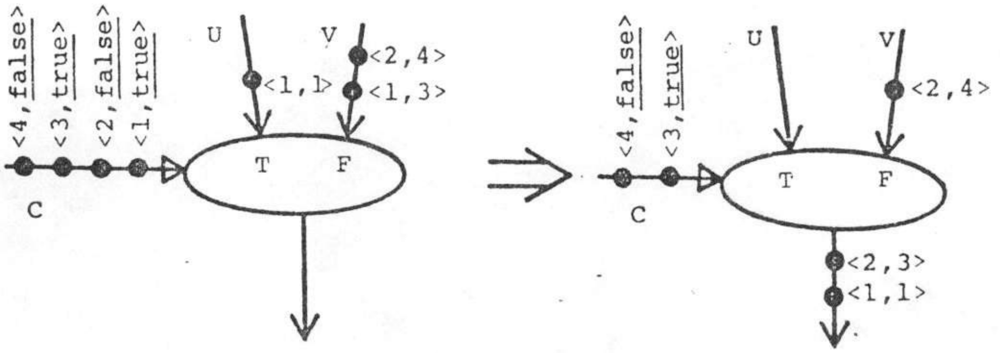


Figure 1c

The merge operator

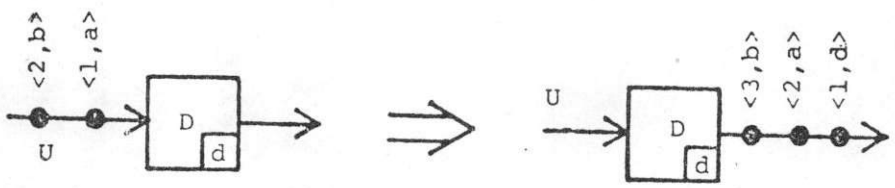


Figure 1d

The D-box operator
(produces initial tokens)

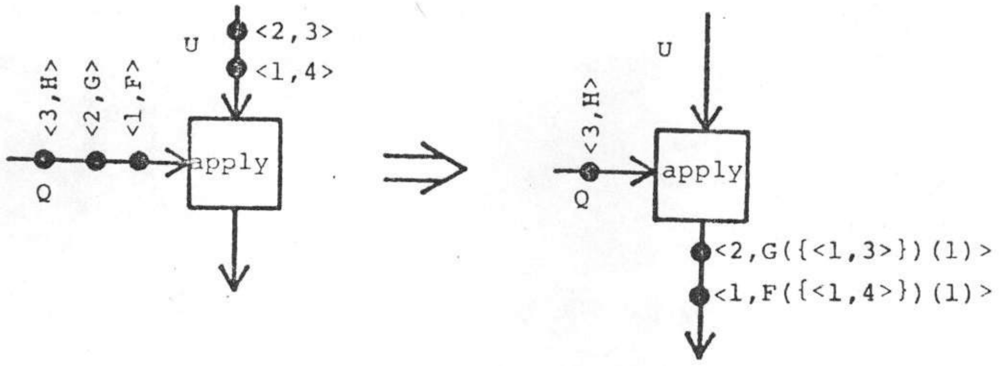


Figure 1e

The apply operator

predicate, gate-if-true (gate-if-false is not shown and is simply the complement of gate-if-true), merge, D, and apply. Dataflow is based upon two principles:

1. An operator fires (produces an output) whenever the inputs required by that operator are present.

2. All operators are functional and produce no side-effects.

Data is carried by tokens that flow along directed arcs connecting the output of one operator to the input of another operator. Thus, in Figure 1a two complete sets of values have arrived at function f, and one value of a third set. A token is represented as an ordered pair <i,v> where i is the logical position of the token relative to the 1st, 2nd, ..., i-1th, ith, i+1th, ... etc. tokens on that same line. The second component v of <i,v> is the data value carried by the token. As can be seen in Figure 1a, function f has fired two times, once for each set of input values. As soon as some token <3,v> arrives on the right-hand input to f, f may fire. Firing causes the input tokens to be destroyed, and if the operator produces an output then a new token is created to carry that output value.

Figure 1b shows the gate-if-true operator whose function is to gate the input data to the output only if the boolean (control) input carries the value true; otherwise no output is produced (but the inputs are still absorbed). Note that the position indicator i in the output tokens follows the order of the input tokens, and does not leave "holes" in the output line's history. Gates may be used to construct a switch operator which in turn may be used to build an if-then-else operator. For example, Figure 2 shows a switch and it

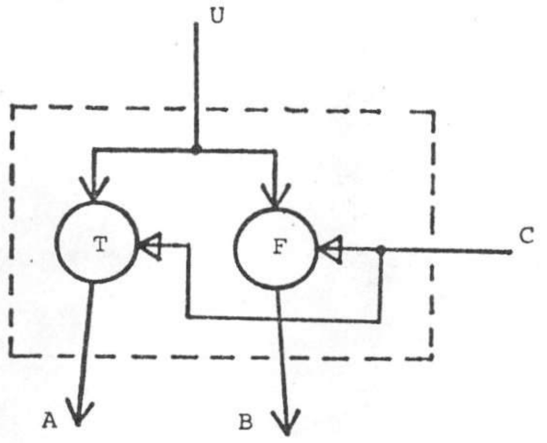


Figure 2

A switch built from
gate-if-true and gate-if-false operators

operates as follows: the ith data input arriving at the switch along line U is <i,u>, and the ith control input <i,c> arrives along line C. The values on lines U and C are sent to both the gate-if-true and gate-if-false operator. This is because there is a fork in each line (represented by the smaller black spot on each of lines U and C); the single token input to the fork is duplicated and two tokens are output by the fork, one token to each destination. Thus, one output line may feed several inputs. In the above example, if c is true then the token <i,u> will appear on line A and no output will appear on line B, or if c is false then <i,u> will appear on line B and no output will appear on line A. A switch thereby directs the flow of data along one of two paths depending on some boolean value. In any dataflow program, we require that any operator expecting boolean values along some line receives only booleans.

A merge operator is shown in Figure 1c and is, in a sense, the inverse of a switch. The purpose of a merge is to join data from two input paths onto a single output path, the order being determined by a third boolean input. Note that each input line has its own history of values and from those two histories and a control history, a third output history is produced.

The D operator serves to introduce an intial token onto a line, and otherwise behaves as the identity function. An example of a dataflow program composed of the above operators is shown in Figure 3. This program has one input line A_1 and one output line X_6.

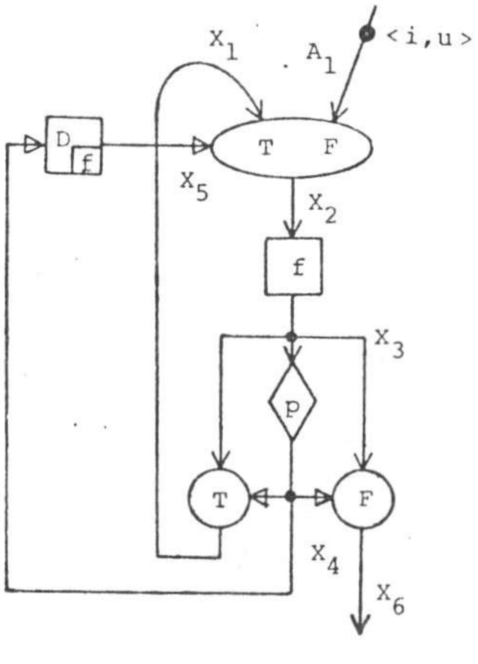


Figure 3

A program in Dennis' dataflow language

The last operator is the apply operator. It accepts two inputs: a dataflow program, and an argument. The value produced is the result of applying the program to the given argument. Only one argument is allowed; multiple arguments may be handled by a suitable encoding of data.

The above has been a brief description of the operators, and one small example of how the operators may be composed to form a program (a term we define more formally below). Of particular importance is that dataflow programs are "functional" since no operator has any side-effects, and no two ouput lines may be connected to a single input line -- the so-called single-assignment restriction. Note that one output may feed many distinct inputs; this produces no side-effects.

3. THE INTERPRETERS

If we assume we have a program comprising a set of m input lines, n operators and thus n output lines (we allow only one output line per operator, but in fact this is no restriction), then there are many possible ways in which the operators could be defined to behave when tokens flow along arcs and arrive at an input. In particular, we can say that each line behaves as a FIFO queue so that an operator may fire using token i from input line L only if token i-1 from L has already been absorbed. That is, the lines behave as pipes down which a sequence of values flows, and the values can never be taken out of sequence. Actually, this is the usual model of dataflow systems [3,5,8,9,10] and we call the underlying interpreter for this model QI meaning Queued Interpreter. Later we will introduce a more asynchronous interpreter UI meaning Unraveling or Unfolding Interpreter which produces results at least as defined as QI, and sometimes more defined. Our interest in UI is due to its asynchrony which, although in most cases produces no more results than QI, it does so with far greater freedom. Then given a suitable architecture [1] that can exploit this asynchrony, the same results can be computed much more quickly.

The model presented in this paper allows us to explicitly capture the notion of asynchrony and thereby make statements about how systems compare with regard to <u>how</u> they compute, and not just what they compute. This is our primary interest.

4. THE MODEL - GENERAL REMARKS AND DEFINITIONS

We model the dataflow language, regardless of interpreter, by considering the history of all tokens produced on a line due to the action of some operator on some histories given to it as input. By varying the definition of the operator as to how it computes outputs for a given input, we can represent each interpreter of interest.

The <u>history</u> of some line L is a set of ordered pairs $<k,v_k>$:

$$L = \{<1,v_1>,<2,v_2>,\ldots,<k,v_k>,\ldots\}$$

where L is a function, i.e., $<k,v_k> \epsilon L \Rightarrow L(k)=v_k$. More precisely,

$$L : J^+ \to D \cup \{\bot\}$$

where $\perp$ is the special undefined value, and where $J^+ = \{0,1,\ldots\}$ is the set of non-negative integers and D is any value domain over which a dataflow program may compute, for example, reals, integers, booleans (denoted true and false here), strings, etc. The range $D\cup\{\perp\}$ of a history function L will be a chain-complete poset with order relation $\sqsubseteq$, where for any $v\in D\cup\{\perp\}$.

$$\perp \sqsubseteq v$$

$$\text{and} \quad v \sqsubseteq v\cdot$$

Note that the value $\perp$ is not in D, and no confusion arises if we define for any history L

$$L(i) = \begin{cases} v \text{ if } \langle i,v\rangle \in L \\ \perp \text{ otherwise} \end{cases}$$

Note that L is a "naturally-extended" function and is thus monotone.

We use the symbol $\subseteq$ to represent a relation on any pair of histories L and L´:

$$L \subseteq L' \text{ iff } (\forall i\in J^+)(L(i) \sqsubseteq L'(i))$$

If we consider a history to be a function defined by a set of <integer,value> ordered pairs, and thus a set of tokens on a line, the $L \subseteq L'$ is exactly token set containment. In particular, there is the totally undefined history represented by the empty set ϕ, where $\phi \subseteq L$ for all histories L.

Let $H = \{L \mid L:J^+ \rightarrow D\cup\{\perp\}\}$ be the set of all possible histories. Histories have certain properties which we need to define in order to conveniently specify the semantics of dataflow operators under various interpreters. The predicate "pr" (pr for "proper") is defined in the domain $H\times J^+$ as

$$pr(L,i) \text{ iff } (\forall j)(1\le j\le i \Rightarrow L(j)\ne\perp)$$

that is, a history is proper up to and including position i if all tokens from 1 through i are in the history; this is equivalent to saying $pr(L,i)$ iff $(\forall j\le i)(\langle i,v\rangle\in L$ for some $v)$. Note that $pr(L,0)$ is always true. We also need two mappings from $H\times J^+$ into J^+:

$$tc(C,i) = \#\{j \mid L(j)=\underline{true} \text{ and } 1\le j\le i\}$$
$$fc(C,i) = \#\{j \mid L(j)=\underline{false} \text{ and } 1\le j\le i\}$$

where #A for some set A means the cardinality of the set A. Thus tc(C,i) is the count of the number of true tokens on line C up to and including the token in the ith position on C. The corresponding statement holds for fc(C,i). As an example in Figure 1b, tc(C,4)=2=tc(C,3), and fc(C,3)=1. At this point we prove some useful facts about histories.

<u>Proposition</u> A - Let A, B, and C be histories, where $A \subseteq B$ and $i \in J^+$. Then

1. $A(i) \neq \bot \Rightarrow A(i)=B(i)$
2. $pr(A,i) \Rightarrow pr(B,i)$
3. $pr(A,i) \Rightarrow tc(A,i)=tc(B,i)$ and $fc(A,i)=fc(B,i)$
4. $C(i)=\underline{true} \Rightarrow tc(C,i)=tc(C,i-1)+1$
   $C(i)=\underline{false} \Rightarrow fc(C,i)=fc(C,i-1)+1$

<u>Proof</u>

1. by the definition of $\subseteq$
2. $pr(A,i) \Rightarrow (\forall j)(1 \le j \le i \Rightarrow A(j) \neq \bot)$
   $\Rightarrow (\forall j)(1 \le j \le i \Rightarrow B(j) \neq \bot)$ by proposition (A.1)
   $\Rightarrow pr(B,i)$ by definition of pr
3. $tc(A,i) = \#\{j \mid A(j)=\underline{true} \wedge 1 \le j \le i\}$
   $= \#\{j \mid B(j)=\underline{true} \wedge 1 \le j \le i\}$ by proposition (A.1)
   $= tc(B,i)$; similarly for fc.
4. $tc(C,i) = \#\{j \mid C(j)=\underline{true} \wedge 1 \le j \le i\}$
   $= \#\{j \mid C(j)=\underline{true} \wedge i \le j \le i-1\} + \#\{i \mid C(i)=\underline{true}\}$
   $= tc(C,i-1)+1$

■

Consider now the single-input DDF program shown in Figure 3. As stated before, input histories will be designated $A_1,\ldots,A_m$, and with no loss in generality we assume each of the n DDF operators in the program has only one output and it is labelled as line L_i for operator τ_i of the program, where

$$\tau_i : H^n \times H^m \to H \qquad 1 \le i \le n$$

Thus each L_i is a history, and a dataflow operator τ_i (function, gate-if-true, merge, etc.) is a functional that maps histories generated internally by the program (the L_j) and histories given as input (the A_k), into an output history L_i.

A dataflow program is a set of n equations on n history variables and m history input parameters where

$$L = \tau[L;A] \tag{4.1}$$

or notationally,

$$L = \tau_A[L]$$

and the symbols stand for vectors of the corresponding symbol.

A <u>meaning</u> of such a program is some fixpoint solution to (4.1). In this paper the least fixpoint of τ_A will be the meaning of a DDF program [9,11,12]. Theorem 2 proves that the least fixpoint is actually the solution that a machine would compute given some rather mild conditions.

The following subsections detail the definitions of the operators as functionals and prove them monotone under interpreter QI and then under UI. To do this precisely, we need one additional semantic notation for "definition by case". The expression

$$(p_1 \rightarrow t_1 \; ; \; p_2 \rightarrow t_2 \; ; \; \ldots \; ; \; p_n \rightarrow t_n \; ; \; t_{n+1})$$

means "if p_1 is true then the result is t_1 otherwise, if p_2 is true then the result is $t_2, \ldots$, if p_n is true then t_n is the result, and if none of $p_1, p_2, \ldots, p_n$ is true then the result is t_{n+1}". We will often nest case definitions, so some of the t_i may themselves be case definitions.

5. THE OPERATORS UNDER QI

Recall that QI is the queued interpreter, meaning that each line behaves as a FIFO queue of tokens where the token $<i,v>$ is the ith token in the queue L. We can describe the behavior of each operator under QI as it accesses these FIFO queues by requiring that pr(L,i) be true if token $<i,v> \in L$ is to be accepted as an input by the operator to produce an output token. If pr(L,i) is not true, then either $<i,v> \notin L$ or some predecessor token in the FIFO queue L is not present. In either case $<i,v>$ cannot take part in the computation until other operators fire and make pr(L,i) true. We do not prove it here, but the operators under QI will never produce any tokens out of FIFO order, that is pr(L,#L) is always true.

5.1 Functions, predicates

These operators are characterized by the fact that the ith set of inputs produces the ith output, and again as for all operators under QI, the FIFO order of token input holds. We let capital letter F name the history (semantic) value of the operator f. Functions and predicates may have n input lines and one output line. Note in the following definition that as more tokens arrive on the various input lines, output appears only in FIFO order and is monotone (we prove this below). We do not distinguish here between program input lines A_i and other lines L_j since to do so would impose additional burdensome notation. We consider the functions and predicates with two inputs only; the other cases are similar.

$$F[U,V] = \bigcup_{i \geq 1} (pr(U,i) \wedge pr(V,i) \rightarrow \{<i,f(U(i),V(i))>\} \; ; \; \phi) \qquad (5.1)$$

Our reason for writing the operators in this fashion rather than a more abstracted function definition, is to mimic as closely as possible actual machine behavior under these operators and the production of output tokens.

<u>Proposition</u> - F[U,V] is monotone
<u>Proof</u> - First we show that F[U,V] is a function if U and V are functions:

$$<i,x> \in F[U,V] \Rightarrow x=f(U(i),V(i))$$
$$\Rightarrow x \text{ is uniquely determined by } f,U,V, \text{ and } i \text{ (since } f,U \text{ and } V \text{ are all functions).}$$

Hence, F can output exactly one token in the ith position of the output queue. Now we prove monotonicity of F:

Let U1,U2,V1, and V2 be histories such that $U1 \subseteq U2 \wedge V1 \subseteq V2$ is true. Let $I1 = \{i \mid pr(U1,i) \wedge pr(V1,i) \text{ is true}\}$.

$$F[U1,V1] = \bigcup_{i \in I1} \{<i,f(U1(i),V1(i))>\} \text{ by equation (5.1)}$$

$$= \bigcup_{i \in I1} \{<i,f(U2(i),V2(i))>\} \text{ by proposition (A.1)}$$

$$\subseteq F[U2,V2] \text{ by equation (5.1)}$$ ∎

5.2 Gates

We show only the gate-if-true operator; corresponding remarks can be made for gate-if-false.

$$\text{GATE-IF-TRUE}[C,U] = \bigcup_{i \geq 1} (pr(C,i) \wedge pr(U,i) \rightarrow (C(i)=\underline{true} \rightarrow \{<tc(C,i),U(i)>\} \; ; \; \phi) \; ; \; \phi) \quad (5.2)$$

That is, gate-if-true will fire for the ith input set from C and U if both are proper up to and including the ith input, and an output token is produced if C(i)=true. In case C(i) is <u>true</u>, then the output produced will be the jth output token where j is the number of <u>true</u> tokens so far absorbed by gate-if-true. No output is produced for any <u>false</u> valued token absorbed on line C. Note that in case either C or U is not proper (gate-if-true cannot yet fire for the ith input set) or if C and U are proper but C(i) is <u>false</u> then the empty set is the result for the ith firing.

<u>Proposition</u> - GATE-IF-TRUE[C,U] is monotone.
<u>Proof</u> - First we show that if C and U are functions, then GATE-IF-TRUE[C,U] is a function.

To establish a contradiction,

$$<j,u_1>,<j,u_2> \in \text{GATE-IF-TRUE}[C,U]$$
$$\Rightarrow (\exists j_1,j_2)(j_1<j_2 \wedge tc(C,j_1)=tc(C,j_2)=j \wedge C(j_1)= C(j_2)=\underline{true})$$

But by Proposition A.4, $tc(C,j_1)<tc(C,j_2)$ which is a contradiction. Hence $<j,u_1>$ and $<j,u_2>$ cannot be distinct tokens output by GATE-IF-TRUE.

To show monotonicity, let C1,C2,U1 and U2 be histories such that $C1 \subseteq C2 \wedge U1 \subseteq U2$

Let $I1 = \{i \mid pr(C1,i) \wedge pr(U1,i) \text{ is true}\}$.

GATE-IF-TRUE[C1,U1]

$$= \bigcup_{i \in I1} (C1(i)=\underline{true} \rightarrow \{<tc(C1,i),U1(i)>\};\phi) \text{ by equation (5.2)}$$

$$= \bigcup_{i \in I1} (C2(i)=\underline{true} \rightarrow \{<tc(C2,i),U2(i)>\};\phi) \text{ by proposition (A.1), (A.2), (A.3)}$$

$$\subseteq \bigcup_{i \in I1} (C2(i)=\underline{true} \rightarrow \{<tc(C2,i),U2(i)>\};\phi) \quad \cup$$

$$\bigcup_{i \in J^+ - I1} (pr(C2,i) \wedge pr(U2,i) \rightarrow (C2(i)=\underline{true} \rightarrow \{<tc(C2,i),U2(i)>\};\phi);\phi)$$

$$= \text{GATE-IF-TRUE}[C2,U2] \text{ by equation (5.2)}$$ ∎

5.3 Merge

$$\text{MERGE}[C,U,V] = \bigcup_{i \ge 1} (pr(C,i) \wedge pr(U,tc(C,i)) \wedge pr(V,fc(C,i)) \rightarrow (C(i)=\underline{true} \rightarrow \{<i,U(tc(C,i))>\}; C(i)=\underline{false} \rightarrow \{<i,V(fc(C,i))>\});\phi) \qquad (5.3)$$

Proposition - MERGE[C,U,V] is monotone.
Proof - First we show that MERGE[C,U,V] is a function if C,U and V are functions. Let $<i,x> \in$ MERGE[C,U,V]. Then x=U(tc(C,i)) if C(i)=true or x=V(fc(C,i)) if C(i)=false. Since C(i) is either true or false but not both, x must be unique. Thus MERGE[C,U,V] is a function.

To show monotonicity, let C1, C2, U1, U2, V1 and V2 be histories such that $C1 \subseteq C2 \wedge U1 \subseteq U2 \wedge V1 \subseteq V2$ let $I1 = \{i | pr(C1,i) \wedge pr(U1,tc(C,i)) \wedge pr(V1,fc(C1,i))$ is true$\}$.

MERGE[C1,U1,V1]

$$= \bigcup_{i \in I1} (C1(i)=\underline{true} \rightarrow \{<i,U1(tc(C1,i))>\}; C1(i)=\underline{false} \rightarrow \{<i,V1(fc(C1,i))>\}) \text{ by equation (5.3)}$$

$$= \bigcup_{i \in I1} (C2(i)=\underline{true} \rightarrow \{<i,U2(tc(C2,i))>\}; C2(i)=\underline{false} \rightarrow \{<i,V2(fc(C2,i))>\}) \text{ by proposition A}$$

$$\subseteq \text{MERGE}[C2,U2,V2] \text{ by equation (5.3)}$$ ∎

5.4 D operator

The D operator introduces an initial token with value b onto a line

$$D_b[U] = \bigcup_{i \ge 1} (pr(U,i) \rightarrow \{<i+1,U(i)>\};\phi) \cup \{<1,b>\} \qquad (5.4)$$

Proposition - $D_b[U]$ is monotone.
Proof - Clear.

5.5 Apply

$$\text{APPLY}[Q,U] = \bigcup_{i\geq 1} (\text{pr}(Q,i) \wedge \text{pr}(U,i) \rightarrow \{<i,\text{apply}(Q(i),U(i))>\};\phi) \quad (5.5)$$

where apply(q,u) is the meaning of the output line of the dataflow program represented by q with input history <1,u>. We require procedure q to be well-behaved [6] so that exactly one token can be produced in the output history of q.

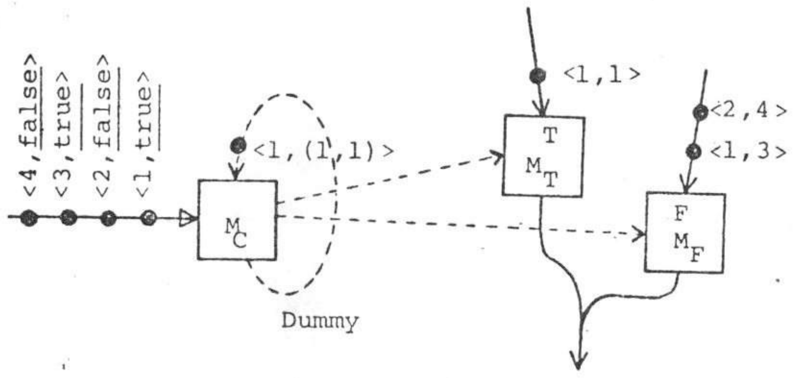
Proposition - APPLY[Q,U] is monotone.
Proof - Just as for the case of functions and predicates.

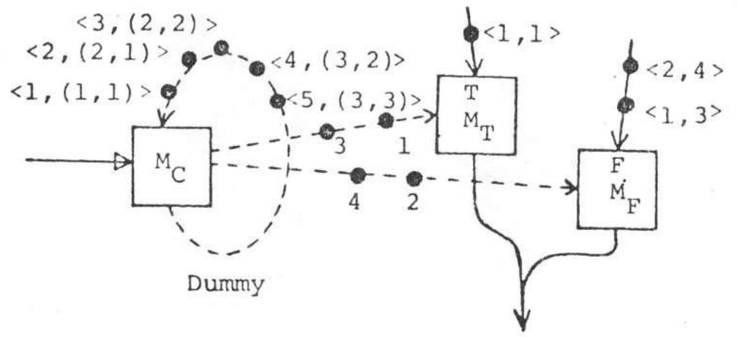
This completes the definition of the operators under QI and the proofs of their monotonicity.

6. THE OPERATORS UNDER UI

We now describe the behavior of the UI interpreter. UI is more asynchronous in its operation than QI, and has the property (shown later) that it may produce even more results than QI. Again, QI is the more usual interpreter and assumes that lines are FIFO queues along which data tokens flow. The position of each token in a queue L corresponds to the relative time at which each token was produced. Time, however, need not be so closely related to queue position. For example, if the ith set of inputs to a function are all present, then regardless of the presence or absence of the i-1th or i+1th set of inputs, the ith output can be calculated. In fact, if each token carried with it a specification of its logical queue position as well as its value (rather than depending upon some physical position in a pipe to specify that information), then by definition the queue position of every token is known and the time at which a token is produced becomes irrelevant. This is precisely what the machine in [1] proposes to do. Theoretically, more results may be produced, but the more important aspect is the increase in asynchrony and potentially concurrent execution, which can be very siginificant. (We have achieved orders of magnitude increases in concurrency of execution with a system based upon the techniques reported here, but where the programmer uses constructs only of the usual arithmetic, if-then-else, and while-do types. By restricting the available constructs, even greater asynchrony can be obtained.)

The following subsections define the behavior of each of the dataflow operators under UI. As an example, consider Figure 4 which shows the internal operation of a merge under UI. Each box may be considered an individual suboperator, where each suboperator behaves just as a function behaves under UI. That is, regardless of the status of the i-1th or i+1th input set to each suboperator, the suboperator may fire the ith output token as soon as the ith input set is present at that suboperator. Thus, Figure 4 shows that merge is three sub-operators called M_C, M_T, and M_F. The dashed lines are just like program lines and have histories but they are internal operator communication lines. The function of M_C is to keep track of which token position from U and from V is to supply the next output token from the merge. Thus information is kept on the line DUM (for Dummy) and is essentially the ordered-value token <i,(tc(C,i), fc(C,i))>. Each time M_C fires, it inputs a token from C and the token from DUM. The input from C says which of lines U and V is to supply an input token, and DUM specifies which position within that line. To cause the appropriate token from U or V to be sent to the merge operator's output, M_C sends a token to M_T or M_F. The value





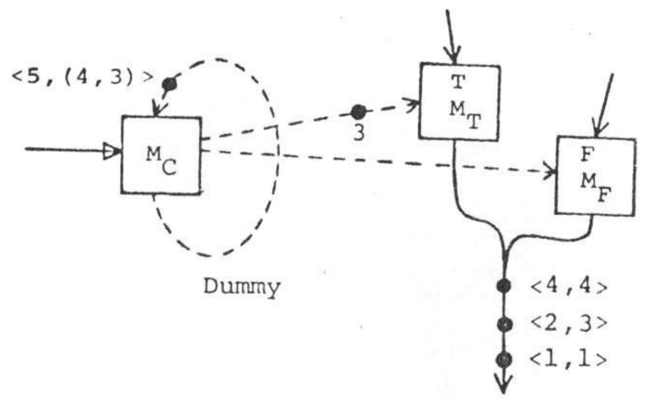


Figure 4

Merge' under the interpreter UI
progresses further than merge under QI for the given input

carried by that token then tells M_T or M_F the output position to be assumed by the token it outputs. Note that M_C must be sequential and only one token exists in DUM at any time, while many tokens may be in transit to M_T and M_F at any instant. In this way, tokens may arrive in arbitrary time order on U and V, and as long as C has defined the appropriate output position, the tokens may be moved by M_T and M_F to the output whenever they arrive.

To denote the interpreter UI, we use the prime mark (´) so F is f according to QI while F´ is f according to UI.

6.1 Functions, predicates - Again we give the case of 2-input functions and predicates.

$$F'[U,V] = \bigcup_{i\geq 1}(U(i)\neq\perp \wedge V(i)\neq\perp \rightarrow \{<i,f(U(i),V(i))>\};\phi) \qquad (6.1)$$

Proposition - F´[U,V] is monotone.
Proof - Similar to the proof for F[U,V] given in section (5.1) provided the set I1 is replaced by the set $\{i \mid U1(i)\neq\perp \wedge V1(i)\neq\perp \text{ is true}\}$.

6.2 Gates

Again we show only the gate-if-true operator. A schematic of the gate-if-true under UI is shown in Figure 5.

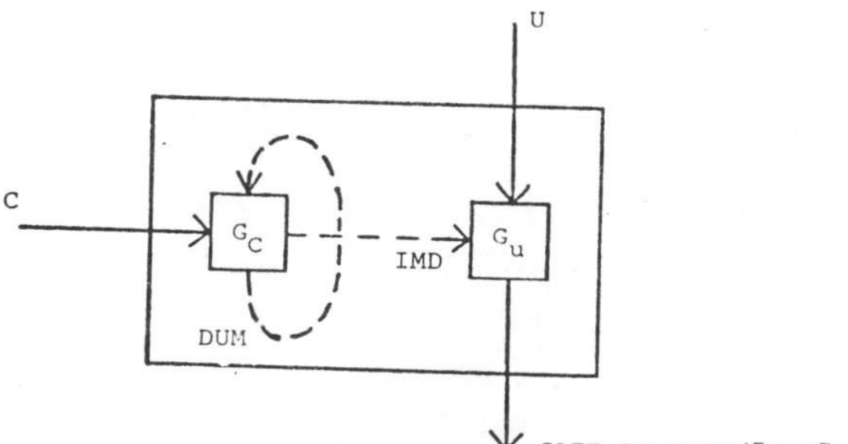


Figure 5

Gate-if-true under interpreter UI

where DUM carries the count of the number of tokens so far output by GATE-IF-TRUE´. Each token on IMD carries an ordered pair as its value. $IMD(i)_1$ selects the first value of the pair and is a copy of C(i). This tells the suboperator G_U whether or not to output U(i). $IMD(i)_2$ selects the second value of the pair, which tells G_U the proper position of the token in the output history.

$$\text{GATE-IF-TRUE}'[C,U] = \bigcup_{i\geq 1} (IMD(i)\neq\bot \wedge U(i)\neq\bot \rightarrow \{<IMD(i)_2, U(i)>\};\phi) \qquad (6.2.1)$$

where, letting $t_i = DUM(i)+(C(i)=\underline{true} \rightarrow 1;0)$ (6.2.2)

$$IMD = \bigcup_{i\geq 1} (DUM(i)\neq\bot \wedge C(i)\neq\bot \rightarrow \{<i,(C(i),t_i)>\};\phi) \qquad (6.2.3)$$

$$DUM = \bigcup_{i\geq 1} (DUM(i)\neq\bot \wedge C(i)\neq\bot \rightarrow \{<i+1,t_i>\};\phi) \cup \{<1,0>\} \qquad (6.2.4)$$

Thus DUM has an initial token in its history, all subsequent members being formed in order iteratively when the corresponding tokens on C arrive. Note that G_C necessarily fires in proper order while G_U may not. In actual operation this sequentiality of G_C is not a great hardship, since predicates in programs are often very short in computation time and waiting for them to arrive in order should not appreciably slow activity.

To prove monotonicity and some other results, we need the following proposition.

Proposition B - Let $i\geq 1$ be an integer; then

1. $DUM(i+1)\neq\bot \Rightarrow pr(C,i)$
2. $pr(C,i) \Rightarrow IMD(i)_2=DUM(i+1)=tc(C,i)$
3. $IMD(i)\neq\bot \Rightarrow pr(C,i)$

Proof

1. By induction

i=1: $DUM(2)\neq\bot \Rightarrow C(1)\neq\bot \Rightarrow pr(C,1)$ by equation (6.2.4)
i=k: $DUM(k+1)\neq\bot \Rightarrow pr(C,k)$ assume
i=k+1: $DUM(k+2)\neq\bot \Rightarrow DUM(k+1)\neq\bot \wedge C(k+1)\neq\bot$ by equation (6.2.4)
$\Rightarrow pr(C,k) \wedge C(k+1)\neq\bot$ by induction hypothesis
$\Rightarrow pr(C,k+1)$ by the definition of pr.

2. By induction

i=1: $pr(C,1) \Rightarrow IMD(1)=(C(1),t_1) \wedge DUM(2)=t_1 \wedge$
$t_1=(C(1)=\underline{true}\rightarrow 1;0)$ since $DUM(1)=0\neq\bot$
$\Rightarrow IMD(1)_2=DUM(2)=t_1=tc(C,1)$
i=k: $pr(C,k) \Rightarrow IMD(k)_2=DUM(k+1)=tc(C,k)$ assume
i=k+1: $pr(C,k+1) \Rightarrow pr(C,k) \wedge C(k+1)\neq\bot$
$\Rightarrow IMD(k+1)=(C(k+1),t_{k+1}) \wedge DUM(k+2)=t_{k+1}$
by induction hypothesis $DUM(k+1)\neq\bot$
$\Rightarrow IMD(k+1)_2=DUM(k+2)=t_{k+1}=$
$DUM(k+1)+(C(k+1)=\underline{true} \rightarrow 1;0)$
by equation (6.2.2)

$\Rightarrow$ $IMD(k+1)_2=DUM(k+2)=tc(C,k)+(C(k+1)=true \rightarrow 1;0)$ by induction hypothesis.

$\Rightarrow$ $IMD(k+1)_2=DUM(k+2)=tc(C,k+1)$ by the definition of tc.

3. Immediate by proposition (B.1) and equation (6.2.3):

$$IMD(k)\neq\bot \Rightarrow DUM(k+1)\neq\bot \Rightarrow pr(C,k) \qquad ■$$

Due to proposition B, IMD can be eliminated from equation (6.2.1). Hence

$$GATE\text{-}IF\text{-}TRUE'[C,U] = \bigcup_{i\geq 1} (pr(C,i) \wedge U(i)\neq\bot \rightarrow \{<tc(C,i),U(i)>\};\phi) \qquad (6.2)$$

Proposition - GATE-IF-TRUE´[C,U] is monotone.
Proof - The proof that GATE-IF-TRUE´[C,U] is a function if C and U are functions is identical to the proof given in section 5.2. The proof of monotonicity of GATE-IF-TRUE given in section 5.2 is valid here too, provided set I1 is replaced by the set $\{i|pr(C1,i) \wedge U1(i)\neq\bot \text{ is true}\}$. ■

6.3 Merge

Figure 3 is a schematic for merge, where $IMD_T(i)$ and $IMD_F(i)$ inform suboperators M_T and M_F, respectively, the output line position for the token U(i) and V(i). A token DUM(i) carries an ordered pair where $DUM(i)_1$ is the next instance of suboperator M_T to be activated, while $DUM(i)_2$ is that of M_F.

$$MERGE'[C,U,V] = \bigcup_{j\geq 1} (IMD_T(j)\neq\bot \wedge U(j)\neq\bot \rightarrow \{<IMD_T(j),U(j)>\};\phi) \cup$$

$$\bigcup_{i\geq 1} (IMD_F(j)\neq\bot \wedge V(j)\neq\bot \rightarrow \{<IMD_F(j),V(j)>\};\phi) \qquad (6.3.1)$$

where as before, we let $t_i=DUM(i)_1+(C(i)=\underline{true} \rightarrow 1;0)$ (6.3.2)

$f_i=DUM(i)_2+(C(i)=\underline{false} \rightarrow 1;0)$ (6.3.3)

$$IMD_T = \bigcup_{i\geq 1} (DUM(i)\neq\bot \wedge C(i)\neq\bot \rightarrow (C(i)=\underline{true} \rightarrow \{<t_i,i>\};\phi);\phi) \qquad (6.3.4)$$

$$IMD_F = \bigcup_{i\geq 1} (DUM(i)\neq\bot \wedge C(i)\neq\bot \rightarrow (C(i)=\underline{false} \rightarrow \{<f_i,i>\};\phi);\phi) \qquad (6.3.5)$$

$$DUM = \bigcup_{i\geq 1} (DUM(i)\neq\bot \wedge C(i)\neq\bot \rightarrow \{<i+1,(t_i,f_i)>\};\phi) \cup \{<1,(0,0)>\} \qquad (6.3.6)$$

We need the following proposition to prove monotonicity.

Proposition C - For integers $i,j\geq 1$

1. $DUM(i+1)\neq\bot \Rightarrow pr(C,i)$
2. $IMD_T(j)=i \Rightarrow pr(C,i) \wedge C(i)=\underline{true} \wedge j=tc(C,i)$
3. $IMD_F(j)=i \Rightarrow pr(C,i) \wedge C(i)=\underline{false} \wedge j=fc(C,i)$
4. $IMD_T(j)=i \Rightarrow (\forall k)(IMD_F(k)\neq i)$

Proof

1. Same as proof of proposition (B.1)
2. $IMD_T(j)\neq\perp$

   $\Rightarrow\ (\exists i)(t_i=j\ \wedge\ C(i)=\underline{true}\ \wedge\ DUM(i)\neq\perp$ by equation (6.3.4)

   $\Rightarrow\ (\exists i)(pr(C,i)\ \wedge\ C(i)=\underline{true}\ \wedge\ j=t_i)$ by proposition (C.1).

   Now by induction we prove that $t_i=tc(C,i)$

   i=1: $t_1 = DUM(1)_1+(C(1)=\underline{true}\rightarrow 1;0)$ by equation (6.3.2)
   $= (C(1)=true \rightarrow 1;0)$ since $DUM(1)=0$
   $= tc(C,1)$ by definition of tc

   i=k: $t_k=tc(C,k)$ assume
   i=k+1: $t_{k+1} = DUM(k+1)_1+(C(k+1)=\underline{true}\rightarrow 1;0)$
   $= t_k+(C(k+1)=\underline{true}\rightarrow 1;0)$ by equation (6.3.6)
   $= tc(C,k)+(C(k+1)=\underline{true}\rightarrow 1;0)$ by induction hypothesis
   $= tc(C,k+1)$ by definition of tc.

hence, $t_i=tc(C,i)$.

Hence

$IMD_T(j)\neq\perp\ \Rightarrow\ (\exists i)(pr(C,i)\ \wedge\ C(i)=\underline{true}\ \wedge\ j=tc(C,i))$

Now we show that i such that the predicate $(pr(C,i)\ \wedge\ C(i)=\underline{true}\ \wedge\ j=tc(C,i))$ is true, must be unique.

Assume $i_1< i_2$ and both i_1 and i_2 satisfy the above predicate. Then

$j=tc(C,i_2)=tc(C,i_2-1)+1$ since $C(i_2)=\underline{true}$
$>tc(C,i_2-1)$

Due to the definition of tc, $(\forall i)(i<i_2\ \Rightarrow\ tc(C,i)\leq tc(C,i_2))$

Hence $j>tc(C,i_2-1)\geq tc(C,i_1)=j$ contradiction.

Therefore $i_1=i_2$ and

$IMD_T(j)=i\ \Rightarrow pr(C,i)\ \wedge\ C(i)=\underline{true}\ \wedge\ j=tc(C,i)$

3. Same as 2.
4. Obvious ∎

IMD_T and IMD_F can be eliminated from equation (6.3.1) due to proposition C.

MERGE´[C,U,V]

$$= \bigcup_{i\geq 1}(pr(C,i)\ \wedge\ C(i)=\underline{true}\ \wedge\ U(tc(C,i))\neq\perp\rightarrow \{<i,U(tc(C,i))>\};\phi)\ \cup$$

$$\bigcup_{i\geq 1}(pr(C,i)\ \wedge\ C(i)=\underline{false}\ \wedge\ V(fc(C,i))\neq\perp\rightarrow \{<i,V(fc(C,i))>\};\phi)$$

$$= \bigcup_{i\geq 1}(pr(C,i)\ \rightarrow\ (C(i)=\underline{true}\ \wedge\ U(tc(C,i))\neq\perp\ \rightarrow\ \{<i,U(tc(C,i))>\};$$
$$C(i)=\underline{false}\ \wedge\ V(fc(C,i))\neq\perp\ \rightarrow\ \{<i,V(fc(C,i))>\};\phi);\phi)$$

(6.3)

Proposition - MERGE´[C,U,V] is monotone.
Proof - Same as the proof for MERGE in section (5.3) provided set I1 is defined as the set
$\{i \mid pr(C1,i) \wedge ((C1(i)=\underline{true} \wedge U(tc(C1,i))\neq\perp) \vee (C1(i)=\underline{false} \wedge V(fc(C1,i))\neq\perp))$ is true$\}$ ∎

6.4 D operator

As under QI, the D operator under UI serves to introduce an initial token; all other tokens pass through.

$$D'_b[U] = \bigcup_{i\geq 1} (U(i)\neq\perp \rightarrow \{<i+1,U(i)>\};\phi) \cup \{<1,b>\} \quad (6.4)$$

Proposition - D´[U] is monotone.
Proof - Clear.

6.5 Apply

$$APPLY'[Q,U] = \bigcup_{i\geq 1} (Q(i),U(i)\neq\perp \rightarrow \{<i,apply(Q(i),U(i))>\} \; ; \; \phi) \quad (6.5)$$

where apply(q,u) is the same as discussed previously in section(5.5).

Proposition - APPLY´[Q,U] is monotone.
Proof - Clear.

7. COMPARISON OF QI AND UI

Consider a function operator f(u) first under QI and then under UI. Let history

$$U = \{<1,u_1>,<3,u_3>,<5,u_5>\}$$

Then $F[U] = \{<1,f(u_1)>\}$, since pr(U,i) is true only for i=1. No more output can be produced by F[U] until $<2,u_2>$ arrives in history U. However, under UI the result is $F'[U] = \{<1,f(u_1)>,<3,f(u_3)>,<5,f(u_5)>\}$ since FIFO order is unnecessary. Whenever $<2,u_2>$ arrives in history U, then $<2,f(u_2)>$ will be output to F´[U]. This is an example of the greater asynchrony evident in UI than in QI, which we model by saying $F[U] \subseteq F'[U]$ for all U, and in particular there are histories V where $F[V] \subsetneq F'[V]$. In actual machine operation this greater asynchrony under UI could very easily manifest itself in greater speeds of execution. As a final example, consider gate-if-true with control input $C = \{<1,\underline{true}>,<2,\underline{true}>,<3,\underline{false}>,<4,\underline{true}>$ and with data input $U = \{<1,u_1>,<3,u_3>,<4,u_4>\}$. The semantics under QI are $GATE\text{-}IF\text{-}TRUE[C,U]=\{<1,u_1>\}$. Under UI, we have $GATE\text{-}IF\text{-}TRUE'[C,U] = \{<1,u_1>,<3,u_4>\}$ and whenever $<2,u_2>$ arrives in history U, it will be output as $<2,u_2>$. Note that the absence of $<2,u_2>$ on input U did not prevent "later" tokens in U from being output under UI. Again we can write $GATE\text{-}IF\text{-}TRUE[C,U] \subseteq GATE\text{-}IF\text{-}TRUE'[C,U]$.

The following propositions prove that each operator under UI is, in general, more asynchronous than under QI; when speaking of operators in general we write this relationship as $OP \subseteq OP'$.

In the following propositions let C,U and V etc. be any histories.

<u>Proposition</u> - $F \subseteq F'$
<u>Proof</u> - As before we prove the case for binary functions.

Let $I = \{i \mid pr(U,i) \wedge pr(V,i) \text{ is true}\}$
and $I' = \{i \mid U(i) \neq \perp \wedge V(i) \neq \perp \text{ is true}\}$.

$$F[U,V] = \bigcup_{i \in I} \{<i, f(U(i),V(i))>\} \qquad \text{by equation (5.1)}$$

$$\subseteq \bigcup_{i \in I'} \{<i, f(U(i),V(i))>\} \qquad \text{since } I \subseteq I'$$

$$= F'[U,V] \qquad ∎$$

Hence $F \subseteq F'$

<u>Proposition</u> - GATE-IF-TRUE $\subseteq$ GATE-IF-TRUE´
<u>Proof</u>

Let $I = \{i \mid pr(C,i) \wedge pr(U,i) \text{ is true}\}$
$I' = \{i \mid pr(C,i) \wedge U(i) \neq \perp \text{ is true}\}$.
$pr(U,i) \Rightarrow U(i) \neq \perp \Rightarrow I \subseteq I'$

GATE-IF-TRUE[C,U]

$$= \bigcup_{i \in I} (C(i) = \underline{true} \rightarrow \{<tc(C,i), U(i)>\}; \phi) \qquad \text{by equation (5.2)}$$

$$\subseteq \bigcup_{i \in I'} (C(i) = \underline{true} \rightarrow \{<tc(C,i), U(i)>\}; \phi) \qquad \text{since } I \subseteq I'$$

= GATE-IF-TRUE´[C,U] ∎

<u>Proposition</u> - MERGE $\subseteq$ MERGE´
<u>Proof</u>

Let $I = \{i \mid pr(C,i) \wedge pr(U,tc(C,i)) \wedge pr(V,fc(C,i)) \text{ is true}\}$
$I' = \{i \mid pr(C,i) \wedge ((C(i)=\underline{true} \wedge U(tc(C,i)) \neq \perp) \vee$
$\qquad (C(i)=\underline{false} \wedge V(fc(C,i)) \neq \perp)) \text{ is true}\}$.

$i \in I \Rightarrow pr(C,i) \wedge U(tc(C,i)) \neq \perp \wedge V(fc(C,i)) \neq \perp$
by definition of I and proposition A.1
$\Rightarrow pr(C,i) \wedge ((C(i)=\underline{true} \wedge U(tc(C,i)) \neq \perp) \vee$
$\qquad (C(i)=\underline{false} \wedge V(fc(C,i)) \neq \perp)$
$\Rightarrow i \in I'$

hence $I \subseteq I'$

MERGE[C,U,V]

$$= \bigcup_{i \in I} (C(i)=\underline{true} \rightarrow \{<i, U(tc(C,i))>\}; C(i)=\underline{false} \rightarrow \{<i, V(fc(C,i))>\}) \qquad \text{by equation (5.3)}$$

$$\subseteq \bigcup_{i \in I'} (C(i)=\underline{true} \rightarrow \{<i, U(tc(C,i))>\}; C(i)=\underline{false} \rightarrow \{<i, V(fc(C,i))>\}) \qquad \text{since } I \subseteq I'$$

= MERGE´[C,U,V] ∎

<u>Proposition</u> - $D_b \subseteq D_b'$
<u>Proof</u> - Clear. ∎

Proposition - APPLY $\subseteq$ APPLY´
Proof - Clear. ∎

This section has shown that OP $\subseteq$ OP´ for all operators in the dataflow language. In fact the vast majority of inputs to any operator OP results in the relation OP $\subsetneq$ OP´. Since for the same input, UI gives results more defined than QI. For example, $F[A] \subset F'[A]$ for $A = \{<1,a_1>,<3,a_3>\}$; UI is less constrained during execution and thus exhibits greater asynchrony.

The following theorem relates the least fixpoint f_p and f_p' of any two interpreters that satisfy the hypothesis. In particular, QI and UI are such a pair of interpreters.

Theorem 1 - If τ and τ' are monotonic functionals and $\tau \subseteq \tau'$, then $f_p \subseteq f_p'$.

Proof - First we show by induction that

$$(\forall i)(\tau^i[\phi] \subseteq \tau'^i[\phi])$$

i=1: $\tau[\phi] \subseteq \tau'[\phi]$ since $\tau \subseteq \tau'$.

i=k: $\tau^k[\phi] \subseteq \tau'^k[\phi]$ assume the induction hypothesis.

i=k+1: $\tau^{k+1}[\phi] = \tau[\tau^k[\phi]]$

$\subseteq \tau'[\tau^k[\phi]]$ since $\tau \subseteq \tau'$

$\subseteq \tau'[\tau'^k[\phi]]$ by induction step and by monotonicity of τ'

$= \tau'^{k+1}[\phi]$

hence $(\forall i)(\tau^i[\phi] \subseteq \tau'^i[\phi])$.

Since $f_p = \text{lub}\{\tau^i[\phi]\}$, for any $j \in J^+$

$f_p(j)=x \Rightarrow (\exists i)(\tau^i[\phi](j)=x)$ by definition of the construction of lub from the sequence $\{\tau^i[\phi]\}$ (proof of the lub lemma, pg. 365, [12]).

Let k = minimum i such that $\tau^i[\phi](j)=x$

$\tau^k[\phi](j)=x \Rightarrow \tau'^k[\phi](j)=x$ since $\tau^k[\phi] \subseteq \tau'^k[\phi]$

$\Rightarrow f_p'(j)=x$ since $f_p'=\text{lub}\{\tau'^i[\phi]\}$.

Hence

$$f_p(j)=x \Rightarrow f_p'(j)=x$$

$$f_p \subseteq f_p'$$ ∎

Theorem 1 implies that for some inputs to a dataflow program, UI may produce answers while QI may not. However, identical results are computed ($f_p=f_p'$) if (1) the program terminates and (2) the program is well-formed*, although UI will still be less constrained than QI

*From [6], meaning that dataflow operators may be combined only to form one-in one-out structured constructs such as if-then-else, etc.

during execution.

Suppose A and B represent two interpreters of a language. We say interpreter B is more asynchronous than interpreter A if for all operators OP and all $X \epsilon H$, $OP_A(X) \subseteq OP_B(X)$. This is justified because in an actual machine X includes the partial histories computed before the fixpoint solution is reached, and since $OP_A \subseteq OP_B$ interpreter B may produce more output (get ahead of) interpreter A during the computation, while the inverse is not possible.

Proposition - Let OP´ represent the interpretation of a DDF operator according to UI, and let OP represent the interpretation according to any other interpreter that operates under the following rules:

R(1). Neither guessing of outputs (either the value or the position in a history), nor any redundant calculation is done,

R(2). No further interpretation is attached to the functions and predicates by the operator.

Then $OP(X) \subseteq OP´(X)$

Proof - In the proof of this proposition, we say that a history A is complete proper if pr(A,#A) is true. Thus, if A has n tokens, and they are in positions 1,2,...,n in the history (i.e. there are no holes in the history) then A is complete proper.

For function, predicates, the D operator, and apply, the result is clear since R(1) requires that we wait until we know the result must be produced. R(2) prevents us from "knowing" future inputs based upon past inputs, so we must wait until all current inputs are defined. Waiting until all current inputs are defined (i.e. $\neq \perp$) is precisely the condition in the definitions of the above operators.

For the gate-if-true (or gate-if-false) operator, assume there is interpreter I such that GATE-IF-TRUE´[C,U] $\subsetneq$ GATE-IF-TRUE_I[C,U] for some histories C and U. Let $<i,v> \in \text{GATE-IF-TRUE}_I[C,U]$ but $<i,v> \notin$ GATE-IF-TRUE´[C,U]. Then certainly, by R(1) and R(2) as argued above, we must have a pair of inputs that produced the token <i,v>, where $C(j)=\underline{true} \wedge U(j)=v$. This is the minimal condition on GATE-IF-TRUE_I. Since the only further condition present on GATE-IF-TRUE´ is pr(C,j), we assume $\neg$pr(C,j) holds but <i,v> is produced under interpreter I.

First, note for any complete proper inputs D and V that GATE-IF-TRUE´[D,K] = GATE-IF-TRUE_I[D,K]. Thus, if we "complete" C to D by adding tokens to C to construct D, then $<i,v> \in \text{GATE-IF-TRUE}_I[C,U] \Rightarrow <i,v> \in$ GATE-IF-TRUE´[D,U].

(There is no need to extend U to V since $U(j) \neq \perp$ is the only condition on U in both interpreters.) Clearly, by definition of GATE-IF-TRUE´, i=tc(C,j)=tc(D,j). But let <k,true> be one of the tokens used to complete C to D, where k<j, and thus tc(C,j)<tc(D,j). Thus <i,v> was an incorrect output by interpreter I, and since k was an arbitrary position in C, the condition pr(C,i) under UI is a necessary condition.

The proof of the proposition for merge is similar to the above. ∎

Since each operator is most defined according to UI, the unravelling interpreter is indeed the most asynchronous interpreter of all interpreters that follow rules R(1) and R(2). According to Theorem 1, if f_p represents the least fixpoint according to some interpreter other than UI and the definitions are expressed as continuous functionals, then $f_p \subseteq f_p'$.

8. MACHINES FOR QI AND UI DO COMPUTE THE LEAST FIXPOINT

We have defined the meaning of dataflow programs in terms of the least fixpoint solution to a system of monotone functionals representing various operators. Machines that have been proposed [1,7] for executing dataflow programs represent components of histories by tokens. The process of executing a program is carried out by generating further tokens at each step based on the tokens present and the available hardware resources. The machine terminates execution when no more tokens can be generated, that is, when it reaches a steady-state. This process of starting the program with empty lines and successively refining the history of every line step by step is very similar to the constructive method of finding the least fixpoint by the first recursion theorem of Kleene [12]. However, the process carried out by a machine may not be identical to the one dictated by Kleene's theorem due to lack of resources and scheduling decisions. Theorem 2 below gives some physically reasonable conditions which guarantee that when a machine reaches the steady-state, the history of all lines will be in accordance with the least fixpoint.

First, we give a definition. Given a functional τ and an input function A, a <u>computation sequence</u> of τ and A is any sequence of functions $\{X^{(k)}\}$ that has the following three properties:

1. $X^{(o)} = \phi$
2. Either for some $s \geq o$

$$X^{(s)} = \tau[X^{(s)},A] \quad \text{and}$$

$$(\forall k<s)\,(X^{(k)} \subsetneq X^{(k+1)} \subseteq \tau[X^{(k)},A])$$

or

$$(\forall k \geq o)\,(X^{(k)} \subsetneq X^{(k+1)} \subseteq \tau[X^{(k)},A])$$

3. $\#(X^{(k+1)} - X^{(k)})$ is finite.

Intuitively, a computation sequence is the sequence of partial results generated by any reasonable machine: (1) a machine that begins with all lines empty, (2) never guesses results and does make progress, and (3) never does an infinite amount of computation in a single step. Such a machine halts only when no operators are enabled (i.e., all operators are in equilibrium). Now we prove a theorem that shows that if such a machine halts then indeed it halts at the least fixpoint of τ_A. Furthermore, we also prove that no matter what path the machines takes (i.e., which of the enabled operators actually fires) they all lead to the same final result. In this sense dataflow systems have the extended Church-Rosser property.

<u>Theorem 2</u>. If $\{X^{(k)}\}$ is a computation sequence of monotonic functional τ and monotonic function A, and if there exists some $s>o$ such that $X^{(s)} = \tau[X^{(s)},A]$

Corollary - Given a monotone functional τ and a monotone function A, if $(\forall k > 0)(X^{(k)} \subsetneq X^{(k+1)} \subseteq \tau[X^{(k)}, A])$ then

$$\mathrm{lub}\{X^{(k)}\} \subseteq \mathrm{lub}\{\tau^k[\phi, A]\}$$

Proof - Included in the proof of Theorem 2 part 2.

ACKNOWLEDGEMENTS

We are extremely grateful to Shirley Rasmussen for typing this paper many times while we kept introducing changes. Without her patience this paper may never have been produced almost on time. The formulation in this version of the paper is slightly different and in our opinion better than the earlier versions; for this we are indebted to George Lucker. We also wish to acknowledge Wil Plouffe and Joseph Goguen who helped us in the earlier stages of this work.

REFERENCES

1. Arvind, K. P. Gostelow, (1977). A Computer Capable of Exchanging Processing Elements for Time, (Proceedings IFIP Congress, Toronto, Canada).

2. Arvind, K. P. Gostelow, (1975). A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture, (Tech Report #72, Information and Computer Science Department, University of California, Irvine).

3. Bährs, A., (1972). Operating Patterns, (An Extensible Model of an Extensible Language) (Symposium Theoretical Programming, Novosibirsk, USSR) pp. 217-246.

4. Chamberlin, D. D. (1971). The 'Single-Assignment' Approach to Parallel Processing, (Proceedings FJCC, AFIPS) pp. 263-269.

5. Dennis, J. B., (1975). First Version of a Data Flow Procedure Language (Computation Structures, Group Memo 93, Project MAC, MIT).

6. Dennis, J. B. and J. B. Fosseen, J. P. Linderman, (1972). Data Flow Schemas, (Symposium on Theoretical Programming, Novosibirsk, USSR) pp. 187-216.

7. Dennis, J. B., D. P. Misunas (1975). A Preliminary Architecture for a Basic Data Flow Processor, (The 2nd Annual Symposium on Computer Architecture, Houston) pp. 126-132.

8. Irani, K. B., (1975). A Proposal for a Programming Language for Parallel Processing (NTIS Tech AD-A009641).

9. Kahn, G., (1974). The Semantics of a Simple Language for Parallel Programming, (Preprints of IFIPS).

10. Kosinski, P. R. (1973). A Data Flow Language for Operating Systems Programming (Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices) Vol. 8, No. 9, pp. 89-94.

11. Kosinski, Paul R. (1976). Mathematical Semantics and Data Flow Programming, (Conference Third ACM Symposium on Principles of Programming Languages) pp. 175-184.

12. Manna, Z., (1974). *Mathematical Theory of Computation*, (McGraw-Hill).

13. Weng, Kung-Song, (1975). Stream-Oriented Computation in Recursive Data Flow Schemas (Project MAC Technical Memorandum 68, MIT).

D-9027
5-10