The Semantics of Asynchrony: The Relationships Between Two Different Interpreters of a Programming Language

Arvind

and

Kim P. Gostelow

DEPARTMENT OF
INFORMATION AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA, IRVINE
IRVINE, CALIFORNIA 92664

The Semantics of Asynchrony: The Relationships Between Two Different Interpreters of a Programming Language

Arvind

and

Kim P. Gostelow

August, 1976

Technical Report, #88
Information and Computer Science Department
University of California, Irvine
Irvine, California 92717

This work was supported by NSF grant MCS76-12460.

Index

Abstract

This report describes the meaning of various data flow operators in terms of relationships between the history of input lines and the history of output lines associated with an operator. Data flow programs are formed by interconnecting data flow operators, and by specifying the program input lines. The history of each line in a program is determined by the history of the input lines of the program. The semantics of a program are defined by a set of fixpoint equations determined by the data flow operators and their interconnections. The least fixpoint solution of such a system of equations is considered, and it is shown that all the operators and all programs under the Unraveling Interpreter are more defined than under the usual Queued Interpreter. However, if only physically meaningful (i.e. proper) inputs are considered then it is shown that both interpreters will arrive at the same least fixpoint solution. Since the Unraveling Interpreter is more defined than the Queued Interpreter, the dynamics of token generation under the Unraveling Interpreter, in general, exhibit more asynchrony than under the Queued Interpreter, even if the inputs to the program are restricted to be physically meaningful.

The Semantics of Asynchrony: The Relationships Between Two Different Interpreters of a Programming Language

1. General Remarks: Let us consider a very simple data flow operator f (e.g., Identity, Square, Constant) in terms of how it transforms an input token to produce an output token. According to Dennis' Data Flow [1,2] language and, in particular, the "queued interpreter" QI associated with that language*, whenever an input token is present function f can execute by absorbing the token holding the value x from line X and producing an output token holding the value y on line Y after some unspecified but finite time. Figure 1 shows function f which computes the value given by

$$y = f(x) \tag{1.1}$$

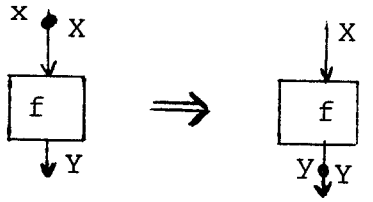

Figure 1. Function f Under Interpreter QI

---

*We have assumed Dennis' interpreter to handle queues of tokens; this is a trival generalization of Dennis' in which a queue of maximum length 1 was assumed.

Even though equation (1.1) relates the value of the input token to the value of the output token, it does not capture the complete semantics of the operator. For example, it does not tell us that the input token is absorbed, nor that the next token on the line X cannot appear unless the previous token has already been absorbed. Due to these observations, it becomes necessary to define every data flow operator in terms of relations between histories of the input lines and histories of the output lines. According to the definition of interpreter QI, tokens appear in sequence on lines. Therefore we can associate a number k with a token if it is the k^{th} token to appear on some line X. The history of line X under QI is then simply a set X of ordered pairs where x_k represents the value of the k^{th} token on line X:

$$X = \{\langle x_1, 1\rangle, \langle x_2, 2\rangle, \ldots, \langle x_k, k\rangle, \ldots\} \qquad (1.2)$$

If the program terminates, the history set associated with every line will be finite.

The situation is slightly different under a new interpreter UI* [2]. Suppose we wish to determine the set

---

*From now on the new interpreter [2] will be called the U-interpreter where U may be read as unraveling or unfolding.

of tokens that are intended for the input X of some statement s, assuming that statement s is a part of procedure P which is called in context u. Then we define the history X under UI to be the set of tokens with prefix u.P.s which are intended for input X of statement s. In the sequel, we ignore the context from which procedure P is called, so the history of input X is represented as

$$X = \{<x_{i_1},i_1>,<x_{i_2},i_2>,\ldots,<x_{i_k},i_k>,\ldots\} \qquad (1.3)$$

Please note that for any set X representing the history of line X, regardless whether under QI or UI, the predicate

$$(<a,i>\epsilon X \text{ and } <b,j>\epsilon X) \Rightarrow i \neq j \qquad (1.4)$$

must be true. Therefore {<a,1>,<b,2>,<c,1>} is not a history.

Let H be the set of all possible histories, including ϕ (the null set). We define a partial order $\subseteq$ on the set H, where $\subseteq$ is set containment.

2. Data flow programs

So far we have only discussed a way to assign meanings to a single data flow operator by means of the histories of input and output lines. The meaning of data flow programs can also be considered in terms of the relationships between the history of the input lines to the program and the history of the output lines from the program. Two programs

are defined to be weakly equivalent iff for all possible inputs they produce the same outputs. However, we will work with the notion of strong equivalence between programs which states that for all possible inputs both programs produce the same histories on all lines. A data flow program defines interconnections between various operators and, hence, relationships between the histories of all the lines. We explain further by giving an example below (Figure 2).

The history associated with line i is represented by X_i. $X_i(k)$ refers to that element of set X_i which has k associated with it as the second value in the ordered pair (e.g., if $X_i = \{<a,1>,<b,3>\}$ then $X_i(3) = b$, $X_i(1) = a$). Note that the value of any history set X_i associated with line i is determined by the value of the token on the program input lines (X_1 in Figure 2), and the interconnection between the various data flow operators. The following algebraic equations represent these relationships for the program of Figure 2.

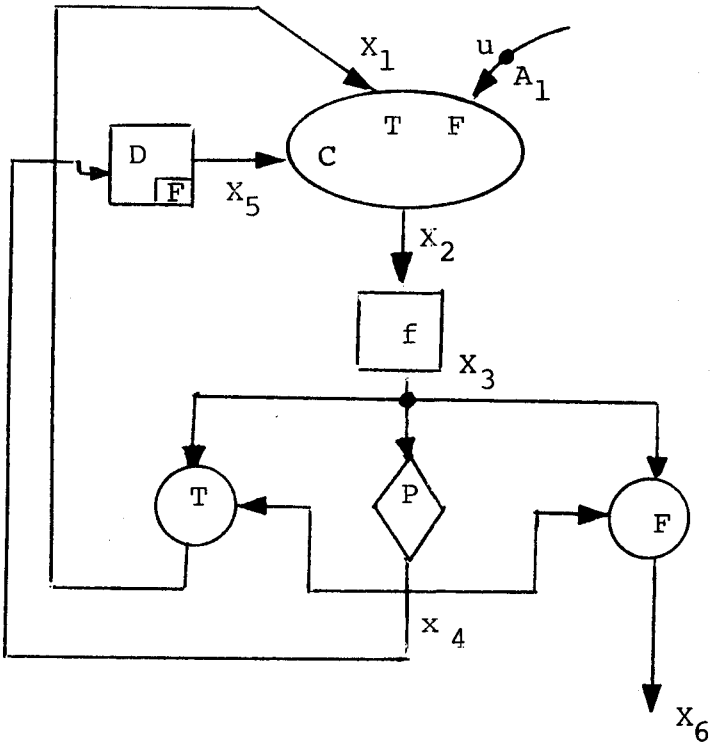

Figure 2. A Data Flow Program

$$A_1(1) = u \qquad \text{(the input)} \tag{2.1}$$

$$X_2(k) = \underline{\text{if}}\ X_5(k)\ \underline{\text{then}}\ X_1(t)\ \underline{\text{else}}\ A_1(m) \tag{2.2}$$

where

t = number of true tokens in $\{X_5(1), X_5(2), \ldots, X_5(k)\}$

m = number of false tokens in $\{X_5(1), X_5(2), \ldots, X_5(k)\}$

$$X_3(k) = f(X_2(k)) \tag{2.3}$$

$$X_4(k) = p(X_3(k)) \tag{2.4}$$

$$X_1(t) = X_3(k)$$

where

$$t = \text{number of true tokens in } \{X_4(1), \ldots, X_4(k)\} \tag{2.5}$$

$$X_5(k) = \underline{\text{if}}\ k=1\ \underline{\text{then}}\ \text{false}\ \underline{\text{else}}\ X_4(k-1) \tag{2.6}$$

$$X_6(t) = X_3(k)$$

where

$$t = \text{number of false tokens in } \{X_4(1), \ldots, X_4(k)\} \tag{2.7}$$

The output is $X_6(1)$.

The above set of equations completely defines the program in Figure 2 once the value of the input token u is specified. Depending upon input token u, the histories of lines X_1 through X_6 will each assume a certain steady state value. Now, if we regard each X_i as a function of input A_1 then the equations (2.1) to (2.7) define a set of fixpoint equations [3] where the functionals are represented in terms

of data flow operators and their interconnections. By the theory of fixpoint equations we know that if all operators are continuous, then the set of equations will yield a unique least fixpoint solution. For the above set of equations the solution will be the history of each line. We propose to determine for any program P the exact relationship between the standard interpreter QI and the new interpreter UI [2] in terms of least fixpoint solutions.

First, we will define the meaning of each type of operator (e.g., merge, gate, function) according to both interpreters in terms of the mapping $H^I \rightarrow H^O$, where I represents the number of input lines to the operator and O represents the number of output lines from the operator. With no loss in generality we assume O=1. The history set X_i associated with output line i will be expressed in terms of a functional τ_i which is a composition of the history variables of all lines and the data flow operators. The histories of the input lines to a dataflow program (i.e., lines without sources) will be designated A_1 to A_m. Let the total number of lines output from all operators in a data flow program be n. We summarize the notation below.

H = a chain-complete partially ordered set of histories.*

---

*Let P be a partially-ordered set and $S \subseteq P$. Then S is a chain if $a,b \in S$ implies $a<b$ or $b<a$, where "<" is the partial order on P. An $x \in P$ is an upper bound of S if $a \leq x$ for all $a \in S$, and x is a least upper bound (lub) for S if $x \leq y$ for all upper bounds y of S. Finally, S is chain-complete if every chain in S has a lub.

$X_i : H^m \rightarrow H$ is a mapping from m input histories to one output history, and is the history associated with the line i, $1 \leq i \leq n$.

$\tau_i : X_1 \times X_2 \times \ldots X_n \rightarrow X_i$ or

$[H^m \rightarrow H]^n \rightarrow [H^m \rightarrow H]$ for $1 \leq i \leq n$.

That is, τ_i is a functional associated with line i and is expressed as a composition of data flow operators and history function variables X_j $(1 \leq j \leq n)$.

A dataflow program P is a set of n equations and m history inputs (A_1 to A_m, such that each $A_i \in H$) where

$$X_i(A_1,\ldots,A_m) = \tau_i[X_1,X_2,\ldots,X_n](A_1,A_2,\ldots,A_m) \quad \text{for } 1 \leq i \leq n \quad (2.8)$$

The meaning of the program P is the least fixpoint solution to the set of equations (2.8).

Before we leave this section we would like to point out that the only physically meaningful initial input histories A_i, $1 \leq i \leq m$ to a program P under interpreter QI are the so-called complete proper histories (i.e., histories of the type $\{\langle u_1,1\rangle,\langle u_2,2\rangle \ldots \langle u_k,k\rangle\}$). In the next section we define such histories rigorously. For the time being it suffices to notice that $\{\langle a,1\rangle,\langle b,2\rangle\}$ is a complete proper history while $\{\langle a,2\rangle\langle b,3\rangle\}$ and $\{\langle a,1\rangle,\langle b,3\rangle\}$ are not.

In Section 7 we will prove that if all the input histories to a program are complete proper histories then both interpreters, QI and UI, produce exactly the same histories on all lines.

The next section contains the definitions of some functions that are used in Sections 4 and 5. They define the semantics of data flow operators.

3. Definitions of some functions

Throughout this section we will assume that B represents the boolean set {true, false}, I represents the set of all positive integers, and R represents the set of values where values may be anything (e.g., reals, booleans, etc.). The functions 3.1-3.7 below are used in characterizing the semantics of data flow operators and programs in Sections 4 and 5.

3.1 value function: $[H \times I \rightarrow R]$

$$value(U,i) = \begin{cases} u & \text{if } \langle u,i\rangle \in U \\ \text{undefined otherwise.} \end{cases} \qquad (3.1)$$

Note that if $U \subseteq U'$ then $value(U,i) = value(U',i)$.

3.2 match function: $[H \times I \rightarrow B]$

$$\text{match}(U,i) = \begin{cases} \text{true if } \langle u,i\rangle \in U \\ \text{false otherwise} \end{cases} \qquad (3.2)$$

Note that if $U \subseteq U'$ then match(U,i) => match(U',i).

3.3 proper-function: $[H \times I \rightarrow B]$

$$\text{proper}(U,i) = \bigwedge_{1 \le j \le i} \text{match}(U,j) \qquad (3.3)$$

Note that if $U \subseteq U'$ then proper(U,i) => proper(U',i).

The next two functions deal with the history associated with control lines.

3.4 true-token-count-function: $[H \times I \rightarrow I]$

$$\text{truecount}(U,i) = \left|\{\langle \text{true},j\rangle \in U \mid j \le i\}\right| \qquad (3.4)$$

3.5 false-token-count-function: $[H \times I \rightarrow I]$

$$\text{falsecount}(U,i) = \left|\{\langle \text{false},j\rangle \in U \mid j \le i\}\right| \qquad (3.5)$$

Again, if $U \subseteq U'$ then

truecount(U,i) $\le$ truecount(U',i) and

falsecount(U,i) $\le$ falsecount(U',i).

If i represents that maximum j such that match(U,j) is true and proper(U,i) is true, then we will call U a complete proper history.

Lastly, we define two functions which deal exclusively with the dummy token within the merge operator. In [2] the dummy token in a merge contains an ordered pair of values

represented as $\langle dum_T, dum_F \rangle$. The history of such dummy tokens will be represented as

$$Dum = \{\ldots, \langle\langle dum_T, dum_F\rangle, i\rangle, \ldots\}$$

Functions truevalue and falsevalue below are defined to extract the values dum_T and dum_F associated with the i^{th} token.

3.6 true-value-of-dummy-function: $[H \times I \rightarrow I]$

$$truevalue(Dum, i) = \begin{cases} dum_T \text{ if } \langle\langle dum_T, dum_F\rangle, i\rangle \in Dum \\ \text{undefined otherwise} \end{cases} \quad (3.6)$$

3.7 false-value-of-dummy-function: $[H \times I \rightarrow I]$

$$falsevalue(Dum, i) = \begin{cases} dum_F \text{ if } \langle\langle dum_T, dum_F\rangle, i\rangle \in Dum \\ \text{undefined otherwise.} \end{cases} \quad (3.7)$$

4. Semantics of DDF operators according to the Queued Interpreter

The meanings of DDF operators in this section and the next section is specified as a function relating the history of input lines to the histories of output lines. Since there are many ways to express the meaning of an operator, there is no way of proving that the equations in these two sections are correct. A reader is encouraged to question the meanings of operators given here and even to give his own meanings to the operators. For this paper, since the functions representing the operators are more important, we will specify whether the meaning of operator f is according to the interpreter QI or is according to the interpreter UI. It should also be noted that whenever new operators are introduced into dataflow, or another dataflow language [6] is used, this same technique can be used to express the meaning of those operators and language.

4.1 Functions and predicates

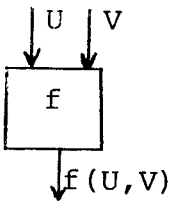

Figure 3. A Function Under QI

$$f(U,V) = \bigcup_{\langle u,i\rangle \in U} (\underline{\text{if}}\ \text{proper}(U,i) \wedge \text{proper}(V,i)\ \underline{\text{then}}\ \{\langle f(u,\text{value}(V,i)),i\rangle\}\ \underline{\text{else}}\ \phi) \qquad (4.1)^*$$

It should be noted that $f(U,V)^{**}$ as defined by equation (4.1) is a monotonic operator, that is, if $U_1 \subseteq U_2$ and $V_1 \subseteq V_2$

---

*Equations (4.1) and some of the equations in the sequel can be written more symmetrically in the following manner

$$f(U,V) = \bigcup_{i \in N} (\underline{\text{if}}\ \text{proper}(U,i) \wedge \text{proper}(V,i)\ \underline{\text{then}}\{\langle f(\text{value}(U,i),\text{value}(V,i)),i\rangle\}\ \underline{\text{else}}\ \phi)$$

where N is the set of natural numbers. However, infinite union of sets poses a problem in proving monotonicity of these functions.

**The notation in equation (4.1) is slightly ambiguous. f on the right hand side denotes the results produced by one application of operator f, while the f on the left hand side operates on the history sets. Unambiguous notation would require using a different symbol (for example F) on the left hand side. We have used the simplified notation throughout the paper on the assumption that an alert reader will not confuse between two f s due to the context.

then $f(U_1,V_1) \subseteq f(U_2, V_2)$. We prove this as follows:

Let $U_2 = U_1 \cup X_1$ where $U_1 \cap X_1 = \phi$

and $V_2 = V_1 \cup X_2$ where $V_1 \cap X_2 = \phi$

From equation (4.1)

$$f(U_2, V_2) = f(U_1 \cup X_1, V_2)$$

$$= \bigcup_{\langle u,i\rangle \in U_1} (\text{if } \text{proper}(U_2,i) \wedge \text{proper}(V_2,i) \text{ then}\{\langle f(u,\text{value}(V_2,i)),i\rangle\} \text{ else } \phi) \cup X_3 \qquad (4.1.1)$$

where

$$X_3 = \bigcup_{\langle x,i\rangle \in X_1} \text{if } \text{proper}(U_2,i) \wedge \text{proper}(V_2,i) \text{ then}\{\langle f(x,\text{value}(V_2,i)),i\rangle\} \text{ else } \phi$$

By the definition (equation 3.3) of proper, we know that if $\text{proper}(U_1,i)$ is true then so is $\text{proper}(U_2,i)$, and similarly $\text{proper}(V_1,i) \Rightarrow \text{proper}(V_2,i)$. Hence, 4.1.1 can be written as

$$f(U_2,V_2) = \bigcup_{\langle u,i\rangle \in U_1} (\text{if } \text{proper}(U_1,i) \wedge \text{proper}(v_1,i) \text{ then } \{\langle f(u,\text{value}(V_2,i)),i\rangle\} \text{ else } \phi) \cup X_4 \cup X_3 \qquad (4.1.2)$$

where X_4 contains $\{\langle f(u,\text{value}(V_2,i)),i\rangle\}$ corresponding to those $\langle u,i\rangle$ for which $\text{proper}(U_2,i) \wedge \text{proper}(V_2,i)$ is true but $\text{proper}(U_1,i) \wedge \text{proper}(V_1,i)$ is false.

In equation (4.1.2) value (V_2,i) will be evaluated only if $\text{match}(V_1,i)$ is true. By the definition (equation (3.1)) of the function value $\text{value}(V_1,i) = \text{value}(V_2,i)$ provided $\text{value}(V_1,i)$ exists. Hence, equation (4.1.2) can be written as

$$\begin{aligned} f(U_2,V_2) &= \bigcup_{\langle u,i\rangle \in U_1} (\text{if } \text{proper}(U_1,i) \wedge \text{proper}(V_1,i) \text{ then } \{\langle f(u,\text{value}(V_1,i)),i\rangle\} \text{ else } \phi) \cup X_4 \cup X_3 \qquad (4.1.3) \\ &= f(U_1,V_1) \cup X_4 \cup X_3 \end{aligned}$$

Hence,

$$f(U_1,V_1) \subseteq f(U_2,V_2) \qquad \text{Q.E.D.}$$

4.2 Gates

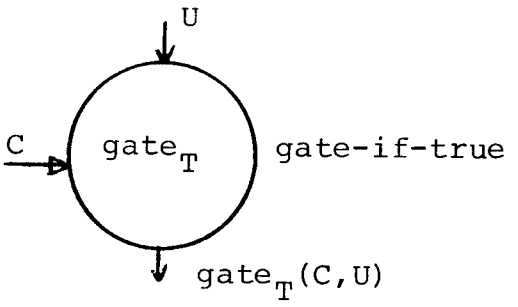

Figure 4. A Gate-if-true Under QI

$$gate_T(C,U) = \bigcup_{<c,i>\in C} (\underline{if}\ proper(C,i) \wedge proper(U,i)\ \underline{then}\ (\underline{if}\ c{=}true\ \underline{then}\ \{<value(U,i),truecount(C,i)>\}\ \underline{else}\ \phi)\ \underline{else}\ \phi) \qquad (4.2)$$

Gate-if-false can be defined in a similar fashion.

4.3 Merge

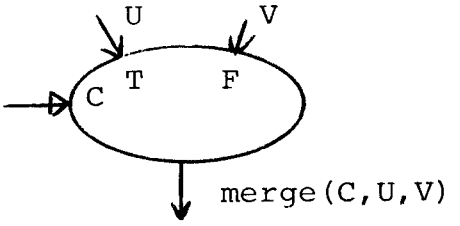

Figure 5. A Merge Under QI

$$merge(C,U,V) = \bigcup_{<c,i>\in C} (\underline{if}\ proper(C,i)\ \underline{then}\ (\underline{if}\ c{=}true\ \underline{then}\ (\underline{if}\ proper(U,truecount(C,i))\ \underline{then}\ \{<value(U,truecount(C,i)),i>\}\ \underline{else}\ \phi)\ \underline{else}\ (\underline{if}\ proper(V,falsecount(C,i))\ \underline{then}\ \{<value(V,falsecount(C,i)),i>\}\ \underline{else}\ \phi))\ \underline{else}\ \phi) \qquad (4.3)$$

4.4 D-box

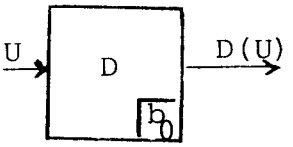

Figure 6. A D-box Under QI

One D-box is required for each initial token present. Note that D-boxes (and hence all initial tokens) are treated as program definition.

$$D(U)=\bigcup_{<u,i>\in U}(\text{if proper}(U,i)\ \text{then}\ \{<u,i+1>\}\ \text{else}\ \phi)\cup\{<b_o,1>\} \qquad (4.4)$$

Discussion:

According to equations (4.1) to (4.4), if inputs are histories, the operators can produce only histories including ϕ, the empty history. History ϕ also represents the undefined history. If a program does not halt then the history of some lines is infinite.

It is easy to check that the gate, merge, and D-box operators are all monotonic with respect to their inputs. The proof is similar to the one given in Section 4.1. Intuitively one can see that the way we defined proper,

match, etc. in Section 3 and the fact that the definition of each operator is given in terms of a union of sets, the right hand side of equations (4.2) to (4.4) can only contain new terms if the inputs are given new tokens.

5. Semantics of DDF operators according to the Unraveling Interpreter

We use a prime mark (′) to indicate that DDF operators are being interpreted according to the interpreter UI.

5.1 Functions and predicates

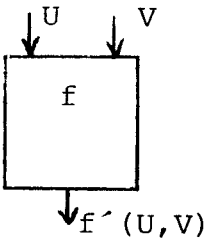

Figure 7. A Function Under UI

$$f'(U,V) = \bigcup_{<u,i>\in U} (\text{if } match(V,i) \text{ then } \{<f(u,value(V,i)),i>\} \text{ else } \phi) \qquad (5.1)$$

This definition of f′ is monotonic with respect to U and V. Further, it is obvious by comparing equations (4.1) and (5.1) that if U and V are complete proper histories then both f and f′ yield the same result. In general, however, f′ is more defined than f, that is

$$f(U,V) \subseteq f'(U,V)$$

5.2 Gates

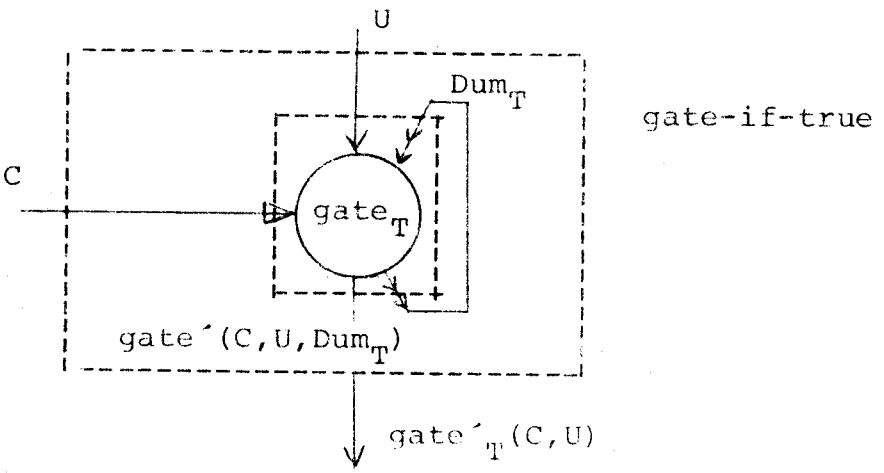

Figure 7. A Gate-if-true Under UI

$$\text{gate}'_T\ (C,U,\text{Dum}_T) = \bigcup_{\langle c,i\rangle \in C}\ (\underline{\text{if}}\ \text{match}(U,i)\wedge\text{match}(\text{Dum}_T,i)\ \underline{\text{then}}\ (\underline{\text{if}}\ c=\text{true}\ \underline{\text{then}}\ \{\langle \text{value}(U,i),\text{value}(\text{Dum}_T,i)\rangle\}\ \underline{\text{else}}\ \phi)\ \underline{\text{else}}\ \phi) \qquad (5.2.1)$$

$$Dum_T(C,U) = \bigcup_{<c,i>\in C} (\text{if } match(U,i) \wedge match(Dum_T,i)$$
$$\text{then (if } c=true$$
$$\text{then}\{<value(Dum_T,i)+1,i+1>\}$$
$$\text{else}\{<value(Dum_T,i),i+1>\})$$
$$\text{else } \phi) \cup \{<1,1>\} \qquad (5.2.2)$$

Discussion: We want to determine the relationship between equations (5.2.1) and (5.2.2), and (4.2). First we define k_U to be the largest k such that proper(U,k) holds. The following lemma shows that Dum_T can only be a complete proper history.

Lemma 1: For a gate-if-true given C and U

$$proper(Dum_T,i) = \begin{cases} true \text{ if } i \leq min(k_C,k_U)+1 \\ false \text{ otherwise} \end{cases}$$

Proof: (by contradiction)

Case 1: Assume that $proper(Dum_T,i) = false$ for some $i \leq min(k_C,k_U)+1$. Further, assume that j is the smallest such i. (Note that $j>1$ since $<1,1> \in Dum_T$ by (5.2.2).) Thus, $match(Dum_T,j)$ is false while $match(Dum_T,j-1)$ is true. Since $j-1 \leq k_C$ and $j-1 \leq k_U$, both match(C,j-1)* and

---

*Instead of saying $<c,i> \in C$ we can write match(C,i) is true.

match(U,j-1) must be true. But according to the definition of Dum_T and the fact that match(C,j-1), match(U,j-1), and match(Dum_T,j-1) are all true, either <value(Dum_T,j-1)+1,j> or <value(Dum_T,j-1),j> must be included in Dum_T by (5.2.2). In either case match(Dum_T,j) is true. Contradiction.

Case 2: Assume that proper(Dum_T,i) = true for some i>min(k_C,k_U)+1. Thus, proper (Dum_T,j) is true provided j≤i. Since i>min(k_C,k_U)+1, match(Dum_T,min(k_C,k_U)+2) must be true. Let us write k for min(k_C,k_U). According to definition (5.2.2) of Dum_T, the only way match(Dum_T,k+2) can be true is if all of match(C,k+1), match(U,k+1) and match(Dum_T,k+1) are true. Since k=min(k_C,k_U), then either k+1>k_C or k+1>k_U or both. Therefore, either match(C,k+1) is false or match(U,k+1) is false or both are false. Contradiction. □

Due to lemma 1, Dum_T is completely determined once C and U are specified. Hence, we may write $gate'_T$(C,U) instead of $gate'_T$(C,U,Dum_T). Also, note that the predicate match(C,i)∧match(U,i)∧match(Dum_T,i) is true if and only if the predicate proper(C,i)∧proper(U,i) is true. Substituting this predicate in equation (5.2.1) we get

$$gate'_T(C,U) = \bigcup_{\langle c,i\rangle \in C} \begin{array}{l} (\underline{if}\ proper(C,i) \wedge proper(U,i) \\ \underline{then}\ (\underline{if}\ c=true\ \underline{then} \\ \quad \{\langle value(U,i), value(Dum_T,i)\rangle\} \\ \qquad\qquad \underline{else}\ \phi) \\ \underline{else}\ \phi) \end{array} \qquad (5.2.3)$$

According to the definition of Dum_T (i.e., equation (5.2.2)) and lemma 1 we note that $value(Dum_T,i)$ is exactly $truecount(C,i)$ provided $value(C,i)$ is true. We clarify by giving an example.

Suppose C = {<T,1>,<T,2>,<F,3>,<F,4>,<F,5>,<T,6>,<F,7>}

Then, i =	1	2	3	4	5	6	7.
value(C,i) =	T	T	F	F	F	T	F
truecount(C,i) =	1	2	2	2	2	3	3
$value(Dum_T,i)$ =	1 ↑	2 ↑	3	3	3	3 ↑	4

From the above table

$$value(Dum_T,i) = \begin{array}{l} \underline{if}\ value(C,i) = true \\ \underline{then}\ truecount(C,i) \\ \underline{else}\ truecount(C,i)+1 \end{array} \qquad (5.2.4)$$

Substituting truecount(C,i) for $value(Dum_T,i)$ under the condition value(C,i)=true in equation (5.2.3) we get

$$gate'_T(C,U) = \bigcup_{\langle c,i\rangle \in C} \begin{array}{l} (\underline{if}\ proper(C,i) \wedge proper(U,i), \\ \quad \underline{then}\ (\underline{if}\ c=true\ \underline{then} \\ \qquad \{\langle value(U,i), truecount(C,i)\rangle\} \\ \qquad\qquad \underline{else}\ \phi) \\ \quad \underline{else}\ \phi) \end{array} \qquad (5.2.5)$$

Equations (5.2.5) and (4.2) are exactly the same. Hence,

$$gate_T(C,U) = gate'_T(C,U)$$

While writing this report a new technique for interpreting gates become apparent. A discussion of the new interpretation and new semantics of the gate operator appear in the appendix. It is proven in the appendix that if UI interprets gates according to the new scheme then

$$gate \leq gate'$$

5.3 Merge

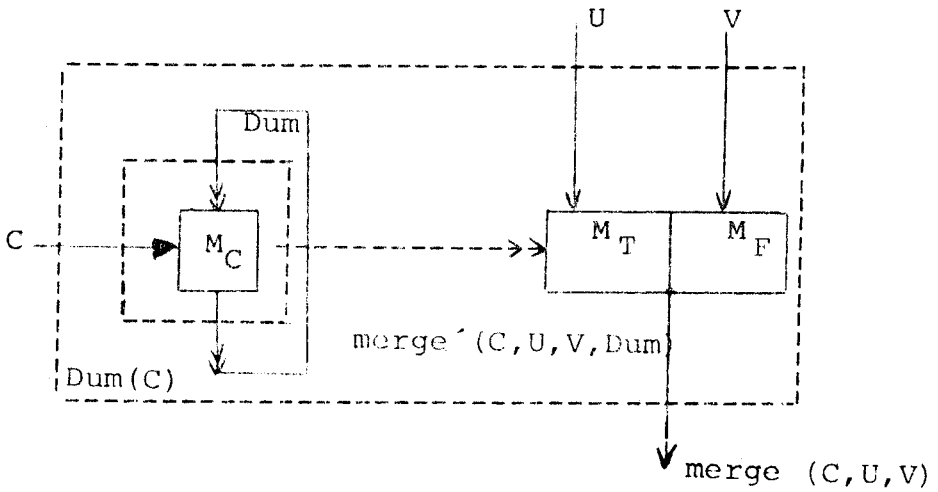

Figure 8. A Merge Under UI

Note that the value of Dum is an ordered pair $<dum_T, dum_F>$.

$$\text{merge}'(C,U,V,Dum) = \bigcup_{\langle c,i\rangle \in C}$$

(if match(Dum,i)
then(if c=true then
(if match(U,truevalue(Dum,i))then
{<value(U,truevalue(Dum,i)),i>}else ϕ)
else
(if match(V,falsevalue(Dum,i)) then
{<value(V,falsevalue(Dum,i)),i>}else ϕ))
else ϕ) (5.3.1)

$$Dum(C) = \bigcup_{\langle c,i\rangle \in C}$$

(if match(Dum,i)then (if c=true then
{<<truevalue(Dum,i)+1,
falsevalue(Dum,i)>,i+1>}
else
{<<truevalue(Dum,i)
value(Dum,i)+1>,i+1>})
else ϕ) {<<1 1>,1>}
(5.3.2)

Discussion: Again we want to compare merge′ and merge. We proceed by eliminating Dum as an argumment to merge′ since Dum is an internal history of the merge statement. Again, let k_U be the largest k such that proper(U,k) holds.

Lemma 2: For a merge statement given input C

$$\text{proper}(Dum,i) = \begin{cases} \text{true} & i \leq k_C+1 \\ \text{false} & \text{otherwise} \end{cases}$$

Proof: (By contradiction)

Case 1: Assume proper(Dum,i) = false for some $i \leq k_C+1$. Further, assume that j is the smallest such i. (Note that $j>1$ since $\langle\langle 1,1\rangle,1\rangle \in$ Dum by (5.3.2).) Then proper(Dum,j-1) = true and proper(Dum,j) = false. Since $j-1 \leq k_C+1$ both match(Dum,j-1) and match(C,j-1) are true. Then due to equation (5.3.2) match(Dum,j) must

be true. Since proper(Dum,j-1) is true and match(Dum,j) is true then proper(Dum,j) is true. Contradiction.

Case 2: Assume proper(Dum.i) = true for some $i>k_C+1$. Thus, proper(Dum,j) is true provided $j\leq i$. Since $k_C+2<i$ both proper(Dum,k_C+2) and match(Dum,k_C+2) must be true. Due to the definition of Dum (equation (5.3.2)) if match(Dum,k_C+2) is true then so must be match(Dum,k_C+1) and match(C,k_C+1). But according to the definition of k_C, proper(C,k_C+1) is false. Since proper(C,k_C) is true proper(C,k_C+1) can be false only if match(C,k_C+1) is false. Contradiction. □

Note that due to lemma 2 the predicate match(C,i)∧match(Dum,i) is true if and only if the predicate proper(C,i) is true. Substituting this in (5.3.1) and noting that since Dum is completely internal to merge´ we can eliminate Dum as an argument to merge´, we arrive at

$$\begin{aligned}\text{merge}'(C,U,V) = \bigcup_{<c,i>\in C} \Big(&\underline{\text{if}}\ \text{proper}(C,i)\ \underline{\text{then}}\ \big(\underline{\text{if}}\ c=\text{true}\\ &\underline{\text{then}}\\ &\quad(\underline{\text{if}}\ \text{match}(U,\text{truevalue}(\text{Dum},i))\ \underline{\text{then}}\\ &\quad\{<\text{value}(U,\text{truevalue}(\text{Dum},i)),i>\}\ \underline{\text{else}}\ \phi)\\ &\quad\underline{\text{else}}\\ &\quad(\underline{\text{if}}\ \text{match}(V,\text{falsevalue}(\text{Dum},i))\ \underline{\text{then}}\\ &\quad\{<\text{value}(V,\text{falsevalue}(\text{Dum},i)),i>\}\ \underline{\text{else}}\ \phi)\big)\\ &\underline{\text{else}}\ \phi\Big)\end{aligned} \qquad (5.3.3)$$

Since proper(Dum,i) is always true provided $\langle c,i\rangle \in C$, we note that truevalue(Dum,i) = truecount(C,i) if value(C,i) is true, and falsevalue(Dum,i) = falsecount(C,i) if value(C,i) is false. Making these substitutions in (5.3.3) we get

$$\begin{aligned}\text{merge}'(C,U,V) = \bigcup_{\langle c,k\rangle \in C} & (\underline{\text{if}}\ \text{proper}(C,i)\ \underline{\text{then}}\ (\underline{\text{if}}\ c=\text{true}\ \underline{\text{then}} \\ & (\underline{\text{if}}\ \text{match}(U,\text{truecount}(C,i))\ \underline{\text{then}} \\ & \{\langle \text{value}(U,\text{truecount}(C,i)),i\rangle\}\ \underline{\text{else}}\ \phi) \\ & \underline{\text{else}} \\ & (\underline{\text{if}}\ \text{match}(V,\text{falsecount}(C,i))\ \underline{\text{then}} \\ & \{\langle \text{value}(V,\text{falsecount}(C,i)),i\rangle\}\ \underline{\text{else}}\ \phi)) \\ & \underline{\text{else}}\ \phi) \qquad (5.3.4)\end{aligned}$$

We would also like to draw attention to the physical behavior of UI. Even though it accepts tokens only in order at the control input, the tokens at the other inputs can be accepted out of order. This implies that merge under UI can produce tokens which are not necessarily in order.

By comparing equations (4.3) and (5.3.4) we notice that the only difference between merge and merge′ is that two occurrences of proper in merge (4.3) have been replaced by two occurrences of match in merge′ (5.3.4). Since proper(W,j) => match(W,j) for any W and j, we conclude that merge′ never contains fewer terms than merge. Hence,

$$\text{merge}(C,U,V) \subseteq \text{merge}'(C,U,V)$$

5.4 D-box

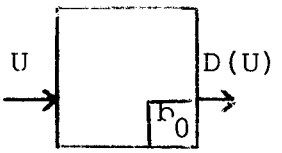

Figure 9. A D-box Under UI

$$D'(U) = \bigcup_{<u,i>\in U} \{<u,i+1>\} \cup \{<b_o,1>\} \qquad (5.4)$$

Clearly $D(U) \subseteq D'(U)$.

Discussion: Again, without proof, we state that operators f',gate',merge' and D' are all monotonic. Intuitively, monotonicity follows once we observe that the right hand side of equations (5.1), (5.2.5), (5.3.4) and (5.4) are defined in terms of union of sets, and increasing the input sets (introducing more input tokens) can only increase the number on terms on the right hand side (produce additional output tokens).

6. Semantics of Procedure Calls (the apply operator)

At least two distinct ways have been suggested for invoking procedures in data flow. According to Dennis [1] an apply operator creates a copy of the called procedure whenever it receives the argument list. In order to distinguish the tokens for the called procedure from the tokens of the calling procedure, a unique color is associated with each procedure invocation. (The activate part of the apply operator changes the color of argument tokens before starting the called procedure.) When the called procedure terminates, it passes the results to the calling procedure by changing back the color of the tokens. Information regarding the change of colors is stored in the terminate part of the apply operator.

Another technique that has been suggested for executing procedures differs from the first one only when more than one argument is allowed in the procedure calls. It has been suggested by Weng [5] that actual substitution of the called procedure code should take place as soon as any of the arguments become available. Of course, some naming or coloring scheme will have to be devised in order not to confuse the calling procedure from the called procedure. The advantage of Weng's scheme is obviously greater asynchrony. Here, we are only interested in the semantics

of procedures, and not the relative merits of these two schemes.

Since the U-interpreter creates a new procedure domain only after all the arguments for invoking a procedure become available*, it executes procedures in a way very similar to Dennis' scheme. One can think of the context part of an activity name (the u in u.P.s.k.) as representing the color of the tokens. In the following we only compare the meaning of procedure calls for Dennis' and Weng's scheme.

Under both schemes the arguments to the called procedure are passed by value. The arguments must be evaluated in the context of the calling procedure, and the called procedure cannot possibly affect the evaluation of arguments. Also, well-behaved data flow procedures, by definition, restore the initial token distribution and thus produce no side-effects (e.g. they exhibit no memory). These restrictions force a strict functional behavior on procedure calls, which in turn means that one invocation of

---

*The U-interpreter can be easily modified to execute procedures according to Weng's scheme.

a procedure cannot interfere with any other invocation of the same procedure. Since Dennis' scheme (as well as the U-interpreter) invokes procedures only after all the arguments become available, the function represented by a well-behaved procedure has to be a naturally extended function* [3] since a procedure cannot produce an output unless all the input tokens arrive. In data flow an absent token (infinite wait) represents the undefined value. When only naturally extended functions are considered, then all computation rules (rules of evaluation) yield the least fixpoint. For the sake of completeness we point out that both the Q and the U-interpreter use the parallel-innermost computation rule [3].

It seems that procedures under Weng's scheme may sometimes produce output even though one or more inputs may be undefined (may not have yet arrived). We conjecture, that if only well-behaved data flow procedures are considered, even Weng's scheme will attach only naturally extended functions as the semantic meaning of a procedure

---

*The value of a naturally extended function is undefined whenever any of its arguments is undefined [3].

call.

Since both the Q-interpreter and the U-interpreter treat procedure application as identical to a function box, based on Sections 4.1 and 5.1 we can write

$$apply \subseteq apply'$$

7. Semantics of data flow programs

In this section let τ_i represent a functional composed of data flow operators whose semantic meanings are specified by Dennis' interpreter QI. Also, let τ'_i be the functional composed of exactly the same data flow operators but whose meanings are specified by the U interpreter. By composition of functions here we mean exactly what is meant by mathematical composition. Instead of functions on the domain of integers,this paper deals with functions defined on the history set H. Physically, operator composition essentially means connecting the output of any data flow operator to the input of one or more data flow operators (including the operator whose output is to be connected). The only restriction is that any input can be connected to at most one output. Hence, convergence of two lines (see Figure 10) is illegal and mathematically meaningless.

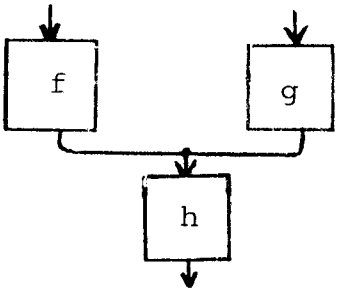

Figure 10. An illegal connection

Since we have already stated that all the data flow operators, under both schemes of interpretation, are monotonic (under $\subseteq$, the set containment relation), then functionals τ_i and τ'_i, which are composed of data flow operators, must be continuous [3]. We have also shown that

$$f \subseteq f' \tag{7.1}$$

$$gate_T = gate'_T \tag{7.2}$$

$$merge \subseteq merge' \tag{7.3}$$

$$D \subseteq D' \tag{7.4}$$

Relations (7.1) to (7.4), along with the fact that all the data flow operators are monotonic imply that $\tau_i \subseteq \tau'_i$.

In Section 2 it was stated that the meaning of a data flow program under QI can be regarded as the least fixpoint solution to a set of n equations where

$$X_i = \tau_i[X_1, X_2, \ldots, X_n] \qquad \text{for } 1 \leq i \leq n$$

We will represent this system of equations in an abbreviated notation as

$$X(A) = \tau[X](A) \tag{7.5}$$

where X represents a vector of history variables, and

τ represents the vector of the functional associated with each line. A is the vector of input histories to the program.

The same program P, when interpreted according to the U-interpreter will be represented as

$$X(A) = \tau'[X](A) \qquad (7.6)$$

Let the solution vector of histories (i.e., the vector of least fixpoints for equations (7.5) and (7.6)) be represented as F_p and F'_p.

7.1 Sufficient Conditions for an implementation to calculate the least fixed point of a dataflow program.

Let us assume that a machine can execute any data flow program written in DDF according to some interpreter. We record the history associated with every line in the program every time any change takes place in the system. We assume without going into the physics of the implementation, that no two changes can take place simultaneously. Let the state of the system at the i^{th} change be represented by $X^{(i)}$. If the machine executes without violating any of the following three conditions, then it will calculate the histories according to the least fixed point of the program. These conditions are

C1. Initally all lines are empty, i.e.,

$$X^{(o)}(A) = \phi$$

C2. At no time does the machine get ahead of the mathematical definition of a function or forget what it has already produced, i.e.,

$$X^{(i)}(A) \subset X^{(i+1)}(A) \subseteq \tau[X^{(i)}](A)$$

C3. The machine is said to stop at time s when the history of all the lines has reached a steady state, i.e.,

$$s = \min_i \{X^{(i)}(A) = \tau[X^{(i)}](A)\}$$

Note that the stopping condition implies that there exists a fixpoint Y of $\tau[X]$ such that $Y(A) = X^{(s)}(A)$

For some inputs, s may be infinitely large implying that the machine does not halt. In such cases history of some line is also infinitely large.

Theorem 1: If Y is defined as follows:

For all $A \in H$

$Y(A)=X^{(s)}(A)$ where $X^{(s)}(A)= \tau[X^{(s)}](A)$

then Y is the least fixpoint of $\tau[X]$.

Proof: (By contradiction). Let $\overset{*}{X}$ be the least fixpoint of $\tau[X]$ and $\overset{*}{X} \neq Y$. Since Y is also a fixpoint of $\tau[X]$,

$\overset{*}{X} \subset Y$. Hence, there must be an input A such that $X(A) \subset Y(A)$. Let B be such an input. Hence

$$\overset{*}{X}(B) \subset Y(B) = X^{(s)}(B)$$

let t be the minimum i such that for some j

$$\overset{*}{X}_j(B) \subset X_j^{(i)}(B)$$

Then by condition C2

$$X_j^{(t)}(B) \subseteq \tau_j[X^{(t-1)}](B)$$

Due to the assumptions about t, $X_k^{(t-1)}(B) \subseteq \overset{*}{X}_k$ for all $k \neq j$ and $X_j^{(t-1)}(B) = \overset{*}{X}_j(B)$ Hence,

$$\tau_j[X^{(t-1)}](B) \subseteq \tau_j[\overset{*}{X}](B)$$

But $\overset{*}{X}$ is a fixpoint of $\tau[X]$, hence

$$\tau_j[\overset{*}{X}](B) = \overset{*}{X}_j(B)$$

Hence, $X_j^{(t)}(B) \subseteq \overset{*}{X}_j(B)$. Contradiction. □

7.2 Relationship between the interpreters QI and UI. Now we state and prove the theorem that gives a relationship between the two interpreters.

Theorem 2: If τ and τ' are continuous functionals and $\tau \subseteq \tau'$ then $F_p \subseteq F'_p$.

Proof: Since τ and τ' are continuous

$$F_p = \text{lub } \{\tau^i[\phi]\} \qquad (7.7)$$

$$F'_p = \text{lub } \{\tau'^i[\phi]\} \qquad (7.8)$$

by Kleene's theorem [3], where ϕ represents the least

defined function* or the empty history and $\tau^{i+1}[\phi] = \tau[\tau^{i}[\phi]]$. Let us define a predicate $\psi \stackrel{\Delta}{=} \tau[X] \subseteq \tau'[X]$. ψ is clearly an admissible predicate [3] because τ and τ' are continuous functionals and ψ has the form of a simple inequality. Hence, if we show by induction that $\tau^{i}[\phi] \subseteq \tau'^{i}[\phi]$ holds for all i then by continuity $\text{lub}\{\tau^{i}[\phi]\} \subseteq \text{lub}\{\tau'^{i}[\phi]\}$ must also hold; then due to equations (7.7) and (7.8), $F_p \subseteq F'_p$ must also be true.

Now we show by induction that $\tau^{i}[\phi] \subseteq \tau'^{i}[\phi]$ is indeed true.

Basis i = 1 $\quad \tau[\phi] \subseteq \tau'[\phi]$ is true because $\tau \subseteq \tau'$

Assume $\quad \tau^{i}[\phi] \subseteq \tau'^{i}[\phi]$

and show $\quad \tau^{i+1}[\phi] \subseteq \tau'^{i+1}[\phi]$.

Now $\tau^{i+1}[\phi] = \tau[\tau^{i}[\phi]] \subseteq \tau[\tau'^{i}[\phi]]$ since τ is monotonic and by the induction step $\tau^{i} \subseteq \tau'^{i}$. But

$\tau[\tau'^{i}[\phi]] \subseteq \tau'[\tau'^{i}[\phi]]$ since $\tau \subseteq \tau'$, or

---

*The use of ϕ here is ambiguous. What is meant by ϕ is the function Ω where $\Omega(A) \stackrel{\Delta}{=} \phi$, for all $A \in H$.

$\tau[\tau'^{i}[\phi]] \subseteq \tau'^{i+1}[\phi]$ by definition of τ'^{i+1};

hence $\tau^{i+1}[\phi] \subseteq \tau'^{i+1}[\phi]$,

hence $F_p \subseteq F'_p$ □

Discussion: One physical interpretation of Theorem 2 is that given an input vector A to a data flow program P the U interpreter may produce some output in addition to the output produced by interpreter QI.

We had briefly indicated towards the end of Section 2 that only physically meaningful histories are those which are complete proper histories (i.e., if for all $\langle u,i\rangle \in U$, proper(U,i) is true then U is a complete proper history). Theorem 3 states the relationship between the two interpreters for a restricted class of input histories.

Theorem 3: If each input history A_i $(1 \leq i \leq m)$ is complete proper, then the interpreters QI and UI will generate proper histories on all lines and $X(A) = \tau[X](A) = \tau'[X](A)$.

Proof: 1. We first prove that for complete proper inputs both interpreters are equal. The proof is based on showing that under both interpreters all operators in DDF produce the same results whenever complete proper histories are considered. This is quite straightforward once we notice that for a complete proper history X,

match(X,i)=proper(X,i).

Functions and predicates: Replace match(V,i) by proper(V,i) in equation 5.1. It reduces to equation (4.1). Hence f = f´.

Gates: We have already shown gate=gate´.

Merges: Replace match(U,truecount(C,i)) by proper(U,truecount(C,i)) and match(V,falsecount(C,i)) by proper (V,falsecount(C,i)) in equation (5.3.4). It reduces to equation (4.3). Hence merge=merge´.

D-box: In equation (4.4) proper(U,i) is always true due to our assumptions. Removing the conditional reduces equation (4.4) to equation (5.4).

Hence, a functional composed of functions, predicates, gates, merges and D-boxes has the same meaning regardless of which interpreter is used if the inputs are restricted to complete proper histories. Hence, $\tau = \tau'$.

According to theorem 2 then $F_p(A) = F'_p(A)$ for the restricted class of complete proper inputs.

2. Now we show that interpreter QI always produces complete proper histories using Scott´s induction rule [4].

A test for complete properness of a history is clearly an admissible predicate. Let $\psi(X_i,\ldots,X_n)$ be true if and only if all $X_i (1 \leq i \leq n)$ are complete proper histories.

Basis $\psi(\phi,\phi,\ldots,\phi)$ is clearly true.

Assume $\psi(X_1,X_2,\ldots,X_n)$ is true and show $\psi(\tau_1[X], \tau_2[X],\ldots, \tau_n[X])$ is also true. Each τ_i must be one of the DDF operators. Therefore, each τ_i is covered by one of the following cases:

Functions and predicates: The right hand side of (4.1) contains one and only one term for each i such that proper(U,i) $\wedge$ proper(V,i) is true. Hence, if <f(u,v),i> is a term on the right hand side then so are <f(u,v),j> for all $j \leq i$. Hence, the right hand side represents a complete proper history.

Gates: Again, by the definition of gate in (4.2) if $\langle w,k \rangle \in gate_T(C,U)$ then match($gate_T$(C,U),j) must be true for all $j \leq k$. Note that if truecount(C,i) is k then truecount(C,i′)$\leq$k for i′<i. Hence, no term below k can be missing from gate(C,V).

Merge: For every i such that proper(C,i) is true a term corresponding to i is produced on the right hand side of equation (4.3). Hence, (4.3) can only produce complete proper histories.

D-boxes: By examination of (4.4) it is obvious that a D-box produces only a complete proper history.

Hence, if each X_i is complete proper then so is each $\tau_i[X]$.

Hence, $\psi(\tau_1[X], \tau_2[X], \ldots, \tau_n[X])$ is true, Hence, $\psi(F_p)$ is true. □

Discussion: Both interpreters produce the same results provided the inputs to the program are complete proper histories. However, theorem 3 only deals with the steady state or the final history of each line. The dynamic behavior, that is, how histories are constructed during the execution of a program will, in general, be quite different under the two interpreters, and UI is, in general, less constrained than QI. The dynamic behavior of the interpreters can also be understood in terms of the conditions for proper implementation stated in section 7.1. If instead of recording changes when they occur, we record them according to a clock which ticks fast enough so as not to miss a change, we will see that the number of clock cycles required to reach the steady state (if it exists) can be less for the U interpreter than the Q interpreter.

Acknowledgements

We are pleased to acknowledge the help of Professor J. Goguen of UCLA in proving theorem 2. Mr. Wil Plouffe's interest in semantics of programs and his critical reading of this paper has been very helpful. It is only due to Ms. Shirley Rasmussen's perseverence and genius that this report could be perpared on a computer inspite of its rigidity.

We would also like to point out that the idea of using the theory of fixpoint equations to define the semantics of parallel programs first occurred to us after reading G. Kahn's [4] paper.

References

[1] Dennis, J. B., "First Version of a Data Flow Procedure Language," Computation Structures, Group Memo 93, Project MAC, MIT November 1973, Revised May 1975.

[2] Arvind, Gostelow, K. P., "A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture", U. C. Irvine, Information and Computer Science Department, Tech Report #72, October 1975.

[3] Manna, Z., Mathematical Theory of Computation (Chapter 5, The Fixpoint Theory of Programming), McGraw-Hill, 1974.

[4] Kahn, G., "The Semantics of a Simple Language for Parallel Programming," Preprints of IFIPS, 1974.

[5] Weng, Kung-Song, "Stream-Oriented Computation in Recursive Data Flow Schemas," Project MAC Technical Memorandum 68, MIT (October 1975).

[6] Kosinski, Paul R., "Mathematical Semantics and Data Flow Programming," Conference Third ACM Symposium on Principles of Programming Languages, 1976, pg. 175-184.

Appendix*

A Note on the Gate Operator

1. General

A gate-if-true operator lets the token pass whenever the corresponding control token is true, otherwise it absorbs the input token.

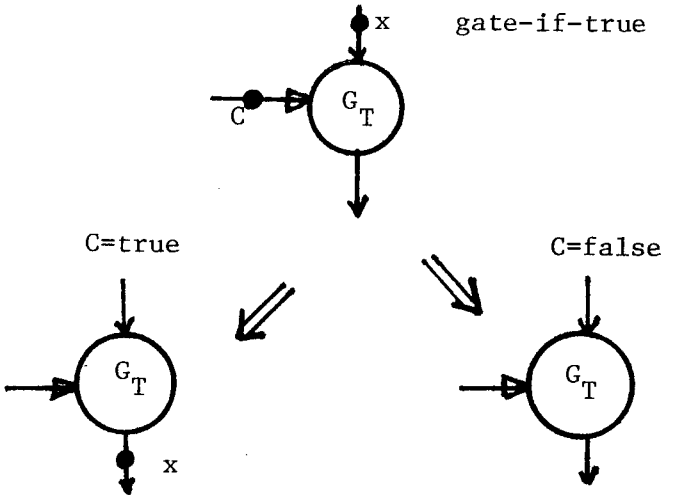

Figure 1

According to the unfolding interpreter [2] the gate must wait for an additional token before executing. This additional token is called the "dummy token" and carries the initiation count for the token to be output. For example, if j true tokens have been received in the first k initiations of a gate operator, then the dummy token will carry the value j+1 for the $k+1^{st}$ initiation of the gate. At each initiation, the gate also outputs an appropriate dummy token for its next execution.

One effect of this scheme of interpretation is that gates exhibit no more asynchrony than the usual scheme of interpretation (à la Dennis).

*This appendix is a reproduction of Data Flow Note 8.

This is due to the chaining effect of the dummy tokens. Consider the following situation. (Since we are interested only in the initiation count part of the activity name, we will show only the initiation count part of the activity names.)

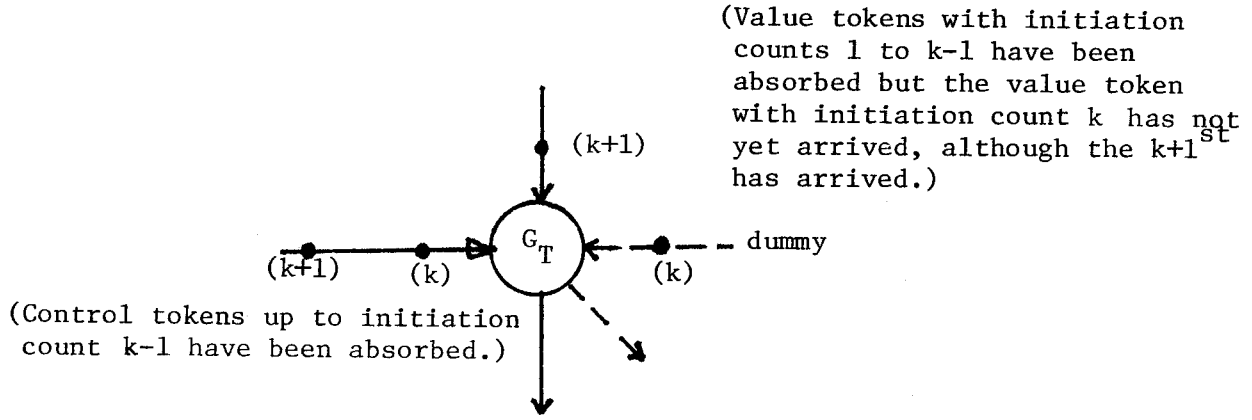

Figure 2

Figure 2 shows a situation where the control tokens with initiation count k and k+1 are available, but on the value side only the token with initiation count k+1 is present. According to [2] the $k+1^{st}$ execution of the gate cannot proceed until the token with initiation count k arrives on the value input. This is unnecessarily restrictive in view of the fact that all information necessary for the $k+1^{st}$ execution of the gate is available even though the k^{th} execution of the gate has not yet taken place. In order to further explain, let us assume that the value of the dummy token for the k^{th} execution is j. The value of the dummy token for the $k+1^{st}$ execution will be j+1 if the k^{th} control token is <u>true</u>, or will be j if the k^{th} control token is <u>false</u>.

The main point being that the moment the value of the control token is known, the value of the dummy token can be determined.

This suggests that the operation of gates can be sped up by splitting the gate into two parts, the control part and the value part. This idea is very similar to the splitting of the merge statements in [2]. The control part will accept a control token and a dummy token and produce as output the usual dummy token and a special "intermediate token" to be directed to the value part. This intermediate token will carry a copy of the control token and the activity number for the output token in case the value token is allowed to pass.

If gates are executed according to the scheme shown in Figure 3, we have proven that only the control parts of a gate need to execute in sequence. The value parts of the gate may execute out of sequence. Since this scheme involves generation of an extra token for each initiation of the gate, it is more expensive in terms of token traffic. It is also worth pointing out that even though this new interpretation of gates can never be slower than the interpretation of gates according to [2], the real benefits will be realized only when control tokens arrive much faster than value tokens. This latter situation is not uncommon when gates are used in the implementation of for-loops or while-loops with easily evaluated predicates.

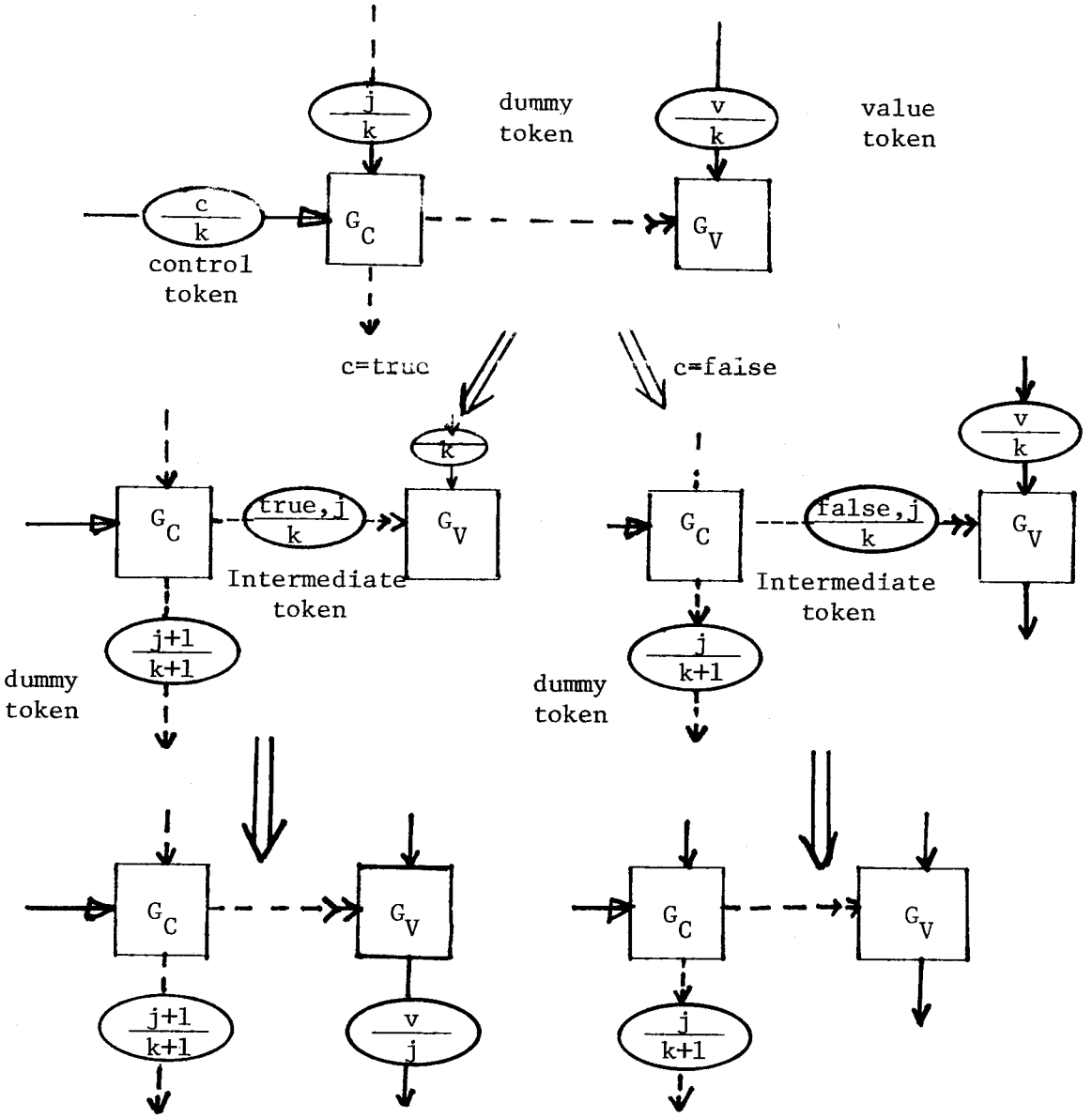

Figure 3: Splitting of a Gate

2. Semantics of the New Interpretation of the Gate Operator

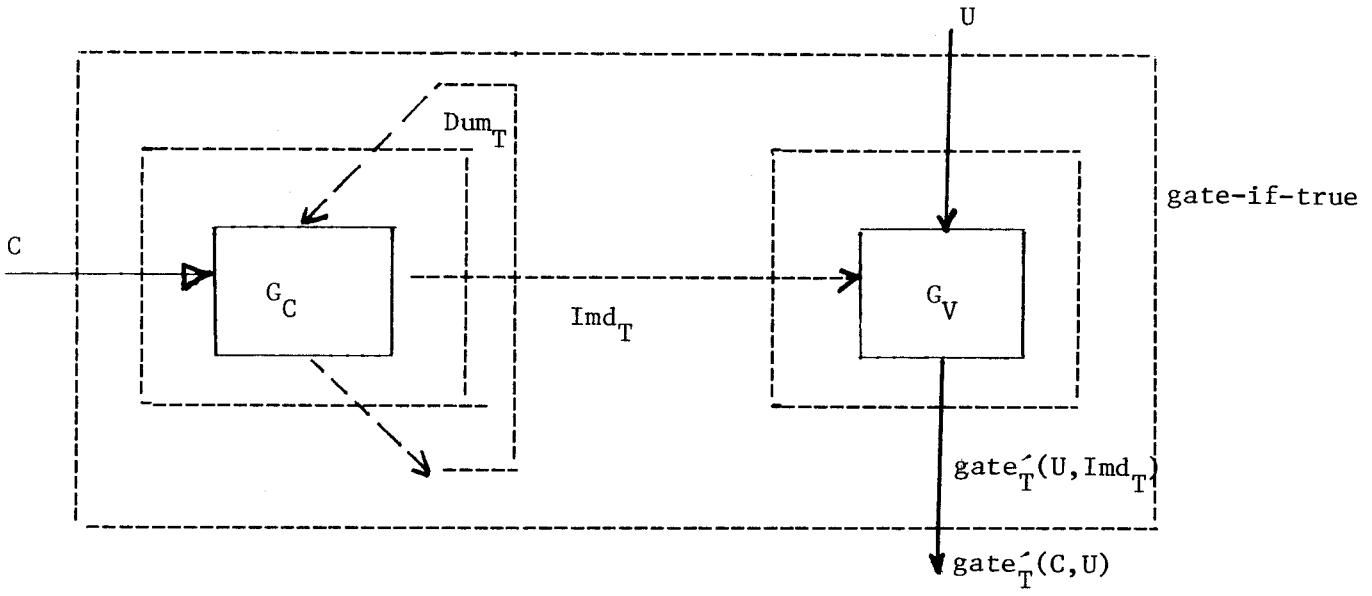

Figure 4

$$\text{Dum}_T(C) = \{<1,1>\} \cup \bigcup_{<c,i>\in C} (\underline{\text{if}}\ \text{match}(\text{Dum}_T, i)\ \underline{\text{then}}(\underline{\text{if}}\ c=\text{true}\ \underline{\text{then}}\{<\text{value}(\text{Dum}_T,i)+1, i+1>\}\ \underline{\text{else}}\{<\text{value}(\text{Dum}_T,i), i+1>\})\ \underline{\text{else}}\ \phi) \tag{2.1}$$

$$\text{Imd}_T(C, \text{Dum}_T) = \bigcup_{<c,i>\in C} (\underline{\text{if}}\ \text{match}(\text{Dum}_T, i)\ \underline{\text{then}}\ \{<<c, \text{value}(\text{Dum}_T, i)>, i>\}\ \underline{\text{else}}\ \phi) \tag{2.2}$$

$$\text{gate}'_T(\text{Imd}_T, U) = \bigcup_{<<c,\text{dum}_T>,i>\in \text{Imd}_T} (\underline{\text{if}}\ \text{match}(U,i)\ \underline{\text{then}}\ (\underline{\text{if}}\ c=\text{true}\ \underline{\text{then}}\{<\text{value}(U,i), \text{dum}_T>\}\ \underline{\text{else}}\ \phi)\ \underline{\text{else}}\ \phi) \tag{2.3}$$

It should be noted that given a C, according to equation (2.1), Dum_T is completely determined,and, similarly, once C and Dum_T are known, Imd_T

is completely determined by equation (2.2). We now state lemma 1 which is used in determining the histories Dum_T and Imd_T. Let k_c represent the largest k such that proper(C,k) is true.

Lemma 1. For a gate-if-true, given C

$$\text{proper}(Dum_T,i) = \begin{cases} \text{true} & \text{if } 1 \le i \le k_c+1 \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{proper}(Imd_T,i) = \begin{cases} \text{true} & \text{if } 1 \le i \le k_c \\ \text{false} & \text{otherwise} \end{cases}$$

The proof of this lemma is quite straightforward, using equations (2.1) and (2.2).

Equation (2.3) shows gate´ to be a function of U and Imd_T. We have already shown that Imd_T is completely known once C is specified. Therefore, we can determine the expression of gate´ as a function of U and C (i.e., $gate'_T(C,U)$, note that both Dum_T and Imd_T are completely internal to the definition of a gate). Due to lemma 1 we know that if $\langle\langle c,dum_T\rangle,i\rangle \in Imd_T^*$ then proper(Imd_T,i) must be true. Hence, we could rewrite equation (2.3) as follows.

$$gate'_T(C,U) = \bigcup_{\langle\langle c,dum_T\rangle,i\rangle \in Imd_T} (\underline{\text{if}}\ \text{match}(U,i) \wedge \text{proper}(Imd_T,i)\ \underline{\text{then}}\ (\underline{\text{if}}\ c=\text{true}\ \underline{\text{then}}\ \{\langle \text{value}(U,i), dum_T\rangle\}\ \underline{\text{else}}\ \phi)\ \underline{\text{else}}\ \phi) \quad (2.4)$$

From equation (2.1) and lemma 1 we assert that value(Dum_T,i) is exactly the number of true tokens in the first i tokens of C provided value(C,i) is true. Hence,

---

*In general, $\langle u,i\rangle \in U$ iff match(U,i) is true.

$$\text{value}(\text{Dum}_T, i) = \underline{\text{if}}\ \text{value}(C,i)=\underline{\text{true}}\ \underline{\text{then}}\ \text{truecount}(C,i)\ \underline{\text{else}}\ \text{truecount}(C,i)+1 \qquad (2.5)$$

However, $\text{proper}(\text{Imd}_T, i)$ is true only if $\text{proper}(C,i)$ is true. Also whenever $\langle\langle c, \text{dum}_T\rangle, i\rangle \in \text{Imd}_T$ then so is $\langle c,i\rangle \in C$ and visa versa. Moreover, dum_T in equation (2.4) is $\text{value}(\text{Dum}_T, i)$. Substituting $\text{proper}(C,i)$ for $\text{proper}(\text{Imd}_T, i)$ and $\text{value}(\text{Dum}_T, i)$ (equation (2.5)) for dum_T in equation (2.4) we get,

$$\text{gate}'_T(C,U) = \bigcup_{\langle c,i\rangle \in C} (\underline{\text{if}}\ \text{match}(U,i) \wedge \text{proper}(C,i)\ \underline{\text{then}}\ \underline{\text{if}}\ c=\text{true}\ \underline{\text{then}}\ \{\langle \text{value}(U,i), \text{truecount}(C,i)\rangle\}\ \underline{\text{else}}\ \phi)\ \underline{\text{else}}\ \phi) \qquad (2.6)$$

We reproduce here the usual semantics of the gate-if-true operator for the sake of comparison.

$$\text{gate}_T(C,U) = \bigcup_{\langle c,i\rangle \in C} (\underline{\text{if}}\ \text{proper}(U,i) \wedge \text{proper}(C,i)\ \underline{\text{then}}\ (\underline{\text{if}}\ c=\text{true}\ \underline{\text{then}}\ \{\langle \text{value}(U,i), \text{truecount}(C,i)\rangle\}\ \underline{\text{else}}\ \phi)\ \underline{\text{else}}\ \phi) \qquad (2.7)$$

From equations (2.6) and (2.7) it is obvious that given a C and a U either

$$\text{gate}_T(C,U) = \text{gate}'_T(C,U)$$

or

$$\text{gate}_T(C,U) \subset \text{gate}'_T(C,U) \quad .$$

Hence, in general,

$$\text{gate}_T \subseteq \text{gate}'_T \quad .$$

3. Monotonicity of the Gate Operator

In this section we show that $gate'_T$, as defined in equation (2.6), is a monotonic operator; that is, given $U_1 \subseteq U_2$ and $C_1 \subseteq C_2$, then

$$gate'_T(U_1,C_1) \subseteq gate'_T(U_2,C_2) \quad .$$

Let $U_2 = U_1 \cup X_1$, where $U_1 \cap X_1 = \phi$ and $C_2 = C_1 \cup X_2$, where $C_1 \cap X_2 = \phi$. Therefore, whenever match(U,i) is true then so is match(U_2,i). Similarly, whenever proper(C_1,i) holds, so does proper(C_2,i).

Also,

$$value(U_1,i) = value(U_2,i)$$

and

$$truecount(C_1,i) = truecount(C_2,i)$$

provided value(U_1,i) and truecount(C_1,i) are defined. Hence, by equation (2.6)

$$gate'_T(C_2,U_2) = gate'_T(C_1,U_1) \cup X$$

where X is some set, possibly empty, determined by the sets X_1 and X_2 in U_2 and C_2.

Therefore,

$$gate'_T(C_1,U_1) \subseteq gate'_T(C_2,U_2) \quad .$$