Massachusetts Institute of Technology

Laboratory for Computer Science

Computation Structures Group Memo 157

A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs

Paul R. Kosinski

[*This paper is to be published in the Proceedings of the 5th Annual Symposium on Principles of Programming Languages, Sponsored by ACM/SIGACT/SIGPLAN.*]

This research was supported by the Computer Sciences Department of the IBM Research Division, the National Science Foundation under grant DCR75-04060, and the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661.

December 1977

A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs

Paul R. Kosinski

IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, New York 10598
&
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Keywords

Data Flow Programming
Denotational Semantics
Non-determinacy

Abstract

Data flow programming languages are especially amenable to mathematization of their semantics in the denotational style of Scott and Strachey. However, many real world programming problems, such as operating systems and data base inquiry systems, require a programming language capable of non-determinacy because of the non-determinate behavior of their physical environment. To date, there has been no satisfactory denotational semantics of programming languages with non-determinacy. This paper presents a straightforward denotational treatment of non-determinate data flow programs as functions from sets of tagged sequences to sets of tagged sequences. A simple complete partial order on such sets exists, in which the data flow primitives are continuous functions, so that any data flow program computes a well defined function.

Introduction

In recent years a new class of programming languages, called data flow languages, has evolved [1,2]. Unlike most programs, the execution of data flow programs is governed solely by the availability of data, both input and computed, rather than by the movement of one or more abstract locuses of control. One of the virtues of data flow programming is that it allows parallelism to be expressed in a natural fashion. Furthermore, the parallelism can be guaranteed determinate, if desired. The expression of parallelism is one of the early reasons researchers were attracted to data flow. However, data flow is now known to have other advantages as well. The two most important are locality of effect and applicative behavior. Applicative behavior means that data flow operators can be characterized as mathematical functions. Locality of effect means that the mathematical equations for a data flow program can be derived simply by conjoining the equations for the various parts of the program in an "additive" manner. Therefore, data flow languages can be analyzed mathematically almost as easily as "toy" applicative languages (eg. pure LISP) but are more powerful in that they provide parallelism and memory.

Concurrent with the rise of data flow programming has been the development of mathematical approaches to the semantics of programs. The success of syntax theory in making precise the syntax of programs led investigators to attempt to describe the semantic behavior programs with equal precision. There are three main approaches to precise semantics: the operational, the axiomatic, and the denotational or functional semantics. The operational approach, based on the notion of an abstract interpreter, is the most intuitive of the three. The denotational approach of Scott and Strachey [3,4] treats the semantic behavior of a program as a function from inputs to outputs, a well known kind of mathematical object. The axiomatic approach of Floyd [5] and Hoare [6], views a program as relating (in the mathematical sense) the "before" state of the abstract machine to its "after" state.

In the denotational approach, each primitive operation in the language is described by associating with it a "semantic function" which it computes. Thus, a sequence of operations computes the function which is the composition of the component operations' functions. If the operations are performed repeatedly, as in a WHILE loop, the composite function is not so easily determined. Such equations can be solved in certain circumstances by means of the Y, or fixed-point, operator. Scott's contribution has been to show that there exist lattices called reflexive domains in which the Y operator can always apply to give the unique minimal fixed-point solution of such equations, and that such domains characterize programming

languages reasonably well. This approach can be used on applicative languages with relative ease since such languages are based on the ideas of functions and their composition.

A program is said to be non-determinate if it does not always yield the same output when given the same input. Non-determinate program behavior is necessary in order to deal with certain real world situations. This classic example of this is the airline reservations system. The last seat on a given flight may be given to different persons, depending on the arrival time of the reservation requests and not merely on the data representing those requests, which is the same whether or not person A gets the seat.

Non-determinate programs are difficult to mathematize in the denotational framework. This is because one must deal with sets of program states rather than the individual states, which adequately characterize determinate programs, and it is difficult to construct a domain whose elements are such sets. Past attempts [7,8] at constructing domains for non-determinate programs have been rather unsatisfactory due to their complexity.

Overview of Data Flow Programming Languages

DFPL, a Data Flow Programming Language [1,9], has the basic mathematical simplicity of applicative languages without most of their drawbacks. Operators in DFPL functionally transform their inputs to their outputs without ever affecting the state of the rest of the program. Since there is no control flow, there is no GOTO; in spite of this, iteration may be programmed as well as recursion. Most significant though, is the fact that unlike ordinary applicative languages, programs may exhibit memory behavior, that is, the current output may depend on past inputs as well as the current input. Memory in DFPL is not primitive but is programmed like other nonprimitive operators. Its effects are local like those of other operators and it does not permeate the semantics of programs.

A DFPL program is a directed graph whose nodes are operators and whose arcs are data paths. Data in DFPL are pure values, either simple like numbers or compound like arrays or records. An operator "fires" when its required inputs are available on its incoming paths. After a unspecified amount of time, it sends its outputs on its outgoing paths. It is not necessary that all inputs be present before an operator fires, it depends on the particular operator. Similarly, not all outputs may be produced by a given firing. Many operators fire only when all their inputs are present, and produce their outputs all at once, they are analogous to subroutines. Some operators produce a time sequence of output values from one input

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

An operator may either be primitive or defined. An operator is defined as network of other operators which are connected by data paths such that certain paths are connected on one end only. These paths are the parameters of the defined operator. A defined operator operates as if its node were replaced by the network which defines it and the parameter paths spliced to the paths which were connected to that node. Recursive operators may be defined. A defined operator for adding complex numbers is given in the appendix.

There are three classes of operators in DFPL: Simple operators, including the usual arithmetic, logical and aggregate operators (eg. construct and select); Stream operators, including the primitive Switch operators (for conditionals and other data routing) and primitive Hold operator (for memory and iteration); and Non-determinate operators, including the primitive Arbiter (for coping with the non-determinate physical world). Simple operators all have the property that they demand all their inputs to fire, whereupon they produce all their outputs. Furthermore, each firing is independent of any past history, that is, the operator is a function from current input to current output.

Stream operators sometimes do not accept/produce all their inputs/outputs, or their current output may depend on past inputs. Thus we can not describe their functional behavior as simply as before (not producing an output is not the same as producing a null output). But we can describe their behavior if we view them as functions from streams (sequences over time) of inputs to streams of outputs. Not all computable functions from sequences to sequences describe Stream operators however; the function must be causal, that is, the operator may never retract some output upon receiving further input.

Non-determinate operators produce any one of a set of output values (according to whim, or in a real implementation, timing considerations) when presented with specified input values. The primitive Arbiter operator, upon which other Non-determinate operators may be based, takes as input two or more streams and produces as output a stream which is the result of merging the input streams in some arbitrary way. Non-determinate operators may be viewed as relations from sequences to sequences, or more profitably, as we shall soon see, as functions from sets of sequences to sets of sequences.

Another approach to the denotational semantics of non-determinate data flow programs, involving partially ordered events instead of sets of sequences (where an event is either the

production or consumption of a datum), has been reported by Keller [10]. However, this report does not provide a complete semantics.

A Brief Mathematical Background

A partially ordered set (poset) is a set with a relation which is reflexive ($A \sqsubseteq A$), transitive ($A \sqsubseteq B$ & $B \sqsubseteq C$ implies $A \sqsubseteq C$) and antisymmetric ($A \sqsubseteq B$ & $B \sqsubseteq A$ implies $A = B$). The relation may not be total, that is, neither $A \sqsubseteq B$ nor $B \sqsubseteq A$ may hold. A quasi-ordered set is the same as a poset without antisymmetry. A chain is a subset of a poset on which the relation is total, that is, either $A \sqsubseteq B$ or $B \sqsubseteq A$. An upper bound of a subset is an element (in the poset, not necessarily in the subset) which every element of the subset is $\sqsubseteq$ to. A least upper bound (Sup) is an upper bound which is $\sqsubseteq$ all other upper bounds. A chain complete poset (cpo) is one in which each chain has a Sup. Since the empty set is a trivial chain, its Sup (called "$\perp$" or "bottom") must exist in a cpo and is $\sqsubseteq$ all elements of the poset.

A function F on a poset is called isotone (or less precisely, monotone) iff for all X and Y, $X \sqsubseteq Y$ implies $F(X) \sqsubseteq F(Y)$. A function F on a cpo is chain continuous (henceforth simply continuous) iff for all chains C in the poset, F(Sup C) = Sup F(C). It may be shown that any continuous function is also isotone. The theorem which results from all this is: any continuous function on a cpo has a minimal fixed-point, that is, there exists an X such that $F(X) = X$ and for all Y such that $F(Y) = Y$, $X \sqsubseteq Y$; Furthermore, X can be found by taking Sup $\{\perp, F(\perp), F(F(\perp)), F(F(F(\perp))), \ldots\}$ which is a chain because $\perp \sqsubseteq F(\perp)$ by definition, and F is isotone since it is continuous.

A Partial Order Suitable for Data Flow

Since determinate operators are adequately characterized as functions from sequences to sequences, the well known partial order on sequences, namely the "prefix" relation is relevant. If the infinite sequences are included, the poset characterized by the prefix relation is chain complete. All the determinate operators of DFPL, if viewed as functions from sequences to sequences, are continuous and therefore isotone in this poset. Therefore, the fixed-point equations resulting from a determinate data flow program can be solved in the cpo of sequences (where $\perp$ is the empty sequence) [11]. In operational terms, an operator is isotone iff it is causal, and an operator is continuous iff it never waits for an infinite sequence of input data before it starts producing output.

Unfortunately, non-determinate operators are best viewed as functions from sets of sequences to sets of sequences. This demands that determinate operators be treated the same so that the domains and codomains of all operators are compatible. Imposing a partial order on sets of sequences has been a frustrating task. For example, Milner's ordering [7] is really only a quasi-order, which means that the fixed-point equations can only be solved to yield a congruence class of sets of sequences. For DFPL at least, such congruence classes have the counter-intuitive property that one class contains two sets which are totally disjoint. This means that certain fixed-point equations can be solved only to the point of saying "you either get this set or that set, and they have *no* elements in common"!

It is possible to obtain a straightforward partial order by considering sets of tagged sequences of data. Each datum in each sequence in the set has associated with it zero or more tags, each of which identifies the sequence of arbitrary decisions made by a non-determinate operator which contributed to the existence of that datum in that sequence. A tag is a sequence of an Arbiter name followed by numbers which denote the decisions made by that Arbiter. Sets of tagged sequences are constrained in the following two ways. First, the tag set of a later datum in a given sequence must be an extension of a tag set of an earlier datum in that sequence, where a tag set T_2 is said to extend T_1 if there is a injection from T_1 to T_2 such that each sequence in T_1 is a prefix of its image. This says that a later datum may never be the result of fewer non-determinate decisions than an earlier datum. Second, no tagged sequence in the set may be a prefix of another in that set (where the prefix demands equality of corresponding tag sets as well as data). This says that no sequence is merely an approximation to another.

Two sets are compared by matching each sequence in the first set with a sequence in the second set such that the first sequence is a prefix of the second sequence. This relation, denoted "$\sqsubseteq$", may be shown to be a true partial ordering of sets of tagged sequences, and the resulting poset is chain complete if infinite sequences and sets are admitted.

To prove that "$\sqsubseteq$" is a partial order on Tagged-sequence-sets, we must prove that it is reflexive, transitive and antisymmetric. Reflexivity is obvious: take the identity map as the injection of Tss_1 to Tss_1. Since any Tagged-sequence is a prefix of itself, we have $Tss_1 \sqsubseteq Tss_1$.

Transitivity is almost as simple. Given an injective map M_1 from Tss_1 to Tss_2, and an injective map M_2 from Tss_2 to Tss_3, we know that the composition $M_2 \circ M_1$ is an injection from Tss_1 to Tss_3. Then, since the prefix relation is transitive, we know that every element in Tss_1 is a prefix of its image (under $M_2 \circ M_1$) in Tss_3. Thus "$\sqsubseteq$" is transitive.

Antisymmetry is the most difficult property to prove; it is the property which the alleged partial orders discussed earlier lack. Let M_1 be an injection from Tss_1 to Tss_2 and M_2 be an injection from Tss_2 to Tss_1. We can immediately conclude that Tss_1 and Tss_2 have the same cardinality and that $M_2 \circ M_1$ is a bijection from Tss_1 to itself. Each element of Tss_1 must be a prefix of its image in Tss_1 under $M_2 \circ M_1$, but due to the constraint on Tagged-sequence-sets, no element can be a prefix of another. Hence the image must be the element itself so $M_2 \circ M_1$ must be the identity. Now we observe that each element of Tss_1 is a prefix of its image in Tss_2 under M_1, and that element in Tss_2 is a prefix of its image in Tss_1 under M_2. But the image under M_2 is the original element in Tss_1, so the element in Tss_2 is equal to the element in Tss_1 by antisymmetry of the prefix relation. Therefore, Tss_2 is equal to Tss_1, and "$\sqsubseteq$" is antisymmetric. □

To show that the partial order "$\sqsubseteq$" is (countable) chain complete, we must show that any countable chain has a Sup. Let $Tss_1 \sqsubseteq Tss_2 \sqsubseteq Tss_3 \sqsubseteq \ldots$ be such a countable chain, and let M_1, M_2 ... be the associated sequence of injective maps which specify the relations (M_1: $Tss_1 \rightarrow Tss_2$, M_2: $Tss_2 \rightarrow Tss_3$, ...). Let S be an element of Tss_N, then the set {S, $M_N(S)$, $[M_N \circ M_{N+1}](S)$, ...} forms a chain under the prefix order and since sequences are chain complete, this set has a Sup which we call Ssup. Call the set of all such Sup's Tss-sup. Since all the M's are injective, each element S of a Tss belongs to exactly one such chain. For each Tss_N, define $Msup_N$ to map each element S into Ssup, the Sup of its chain. Then we have that $Msup_N$: $Tss_N \rightarrow$ Tss-sup is an injective map which establishes that $Tss_N \sqsubseteq$ Tss-sup. But N was arbitrary, so Tss-sup is an upper bound for the chain of Tss's.

If there were another upper bound, call it Tss-ub, for the chain of Tss's which was strictly less than Tss-sup, then there would be an element S-ub in Tss-ub which was a strict prefix of an element S-sup of Tss-sup, or there would be an element in Tss-sup which had no prefix in Tss-ub. In the first case, S-ub would be an upper bound of some chain, but then S-ub is a strict prefix of S-sup, contradicting the fact that S-sup was the Sup of that chain. In the second case, there would be a chain of elements from the Tss's which had no Sup in Tss-ub, hence Tss-ub could not even be an upper bound. Therefore, we may conclude that Tss-sup is indeed the Sup of the Tss's. □

It remains to be shown that Tss-sup satisfies the extra conditions on Tagged-sequence-sets: namely, that no Tagged-sequence is a strict prefix of another, and that within an Tagged-sequence, the Tag-set on a later item in the Tagged-sequence must extend the Tag-set on an earlier item. We prove these additional properties by contradiction.

If one Tagged-sequence, Ts_1, were a strict prefix of another, Ts_2, then all the elements of the chain of which Ts1 was the Sup would be in the chain of Ts_2, hence Ts_1 could not be their Sup. □

If the Tag-set extension property were not obeyed, then there would exist a Tagged-sequence Ts-sup in Tss-sup such that Tag-set(Ts-sup_K) did not extend Tag-set(Ts-sup_J), where $J \leq K$. But, since Tss-sup is the Sup of its chain of Tss's, there would exist some Tss_N which contained a Tagged-sequence Ts a prefix of Ts-sup such that Ts_J = Ts-sup_J and Ts_K = Ts-sup_K contradicting the Tag-set extension property assumed for the Tss's. □

Therefore the Tss-sup is a proper Tagged-sequence-set and is the Sup of the Tss's, which means that the set of Tagged-sequence-sets is a complete poset.

Behavior of Determinate Operators

Any determinate operator, whose functional behavior on simple data sequences is known, may be extended to a function on sets of tagged sequences, but *not* in the obvious way of applying the operator to all possible tuples of sequences in the Cartesian product of the input sets and producing an ouput set whose size is that of that Cartesian product. The problem with this obvious approach is that the operator may be applied to data which could never coexist during actual execution because they were the result of contradictory decisions of the same Arbiter.

The proper extension is as follows: "execute" the operator on each tuple of input sequences in the Cartesian product of the input sets, letting it consume an input datum whenever it wishes and produce an output datum whenever it wishes. However, while doing this, join the tag set associated with the input datum with an accumulating tag set (initially the empty set), where joining two tag sets is done by taking their union and deleting any tags which are prefixes of other tags. Furthermore, whenever an output datum is produced, it is tagged with the current value of the accumulating tag set. The execution of the operator is stopped, before producing any further output, whenever an inconsistent tag set is accumulated. An inconsistent tag set is one which has two tags with the same Arbiter name but with contradictory decisions. This rule assures that the operator's function is never applied to input data which could never co-exist because they arose from different decision sequences of some non-determinate operator. After each tuple is processed in this way, the output sequence is put into the output set, and prefixes eliminated.

Behavior of Non-determinate Operators

The only primitive non-determinate operator is the Arbiter which, viewed as a function from sequences to sets of sequences, produces the set of all possible ways of merging the input sequences such that each datum is tagged by the unique name of the Arbiter (which just tells which Arbiter in the program it is) and the sequence of decisions made so far. For example, if the input sequences $\langle A,B\rangle$ and $\langle C,D\rangle$ were merged by the Arbiter named "α", the output set would be:

$$\{\ \langle A_{\alpha 0},B_{\alpha 00},C_{\alpha 001},D_{\alpha 0011}\rangle,\ \langle A_{\alpha 0},C_{\alpha 01},B_{\alpha 010},D_{\alpha 0101}\rangle,$$
$$\langle A_{\alpha 0},C_{\alpha 01},D_{\alpha 011},B_{\alpha 0110}\rangle,\ \langle C_{\alpha 1},A_{\alpha 10},B_{\alpha 100},D_{\alpha 1001}\rangle,$$
$$\langle C_{\alpha 1},A_{\alpha 10},D_{\alpha 101},B_{\alpha 1010}\rangle,\ \langle C_{\alpha 1},D_{\alpha 11},A_{\alpha 110},B_{\alpha 1100}\rangle\ \}.$$

Viewed as a function from sets of tagged sequences to sets of tagged sequences, the Arbiter is extended like any determinate operator, except that the accumulating tag set always has a generated tag in it which tells the sequence of decisions made so far by the Arbiter (it is initially just the Arbiter's name), and all possible merges are generated in parallel.

Proof of Continuity

To prove that the extended determinate operators are isotone, we first prove a lemma concerning such extensions in general. Let F be a function which maps Tagged-sequences × Tagged-sequences → Tagged-sequences, and call its extension F*. Then define F* as follows (where Tss_i is a Tagged-sequence-set, and Tsa_i and Tsb_i are Tagged-sequences):

$$F^*(Tss_1,Tss_2,\ldots) =$$
$$\{\ F(Tsa_1,Tsa_2,\ldots)\ \textit{not-a-prefix-of}\ F(Tsb_1,Tsb_2,\ldots)\ \mid$$
$$Tsa_1,Tsb_1 \in Tss_1\ \&\ Tsa_2,Tsb_2 \in Tss_2\ \&\ \ldots\ \}$$

We now show that if $Tss_1 \sqsubseteq Tssx_1$, then $F^*(Tss_1,Tss_2,\ldots) \sqsubseteq F^*(Tssx_1,Tss_2,\ldots)$. Pick an arbitrary $Ts \in F^*(Tss_1,Tss_2,\ldots)$, then there exist $Ts_1 \in Tss_1$ and $Ts_2 \in Tss_2$ such that $Ts = F(Ts_1,Ts_2,\ldots)$. Since $Tss_1 \sqsubseteq Tssx_1$, there exists $Tsx_1 \in Tssx_1$ such that Ts_1 is a prefix of Tsx_1, which implies that Ts is a prefix of $F(Tsx_1,Ts_2,\ldots)$ since F is isotone. But either $F(Tsx_1,Ts_2,\ldots) \in F^*(Tssx_1,Tss_2,\ldots)$ or $F(Tsx_1,Ts_2,\ldots)$ was discarded by the "*not-a-prefix-of*" and there exists $Tsx \in F^*(Tssx_1,Tss_2,\ldots)$ such that $F(Tsx_1,Ts_2,\ldots)$ is a prefix of Tsx. Therefore Ts is a prefix of Tsx, but Ts was arbitrary, so for all $Ts \in F^*(Tss_1,Tss_2,\ldots)$ there exists $Tsx \in F^*(Tssx_1,Tss_2,\ldots)$ such that Ts is a prefix of Tsx, and hence (since the sets are prefix-

reduced) $F^*(Tss_1,Tss_2,...) \subseteq F^*(Tssx_1,Tss_2,...)$. □

Now we may prove that such isotone extensions are also continuous. Let Tss_N be a chain whose Sup is Tss. Let F be a function on Tagged-sequences which is continuous and thus isotone. Consider the sequence of image sets $F(Tss_1)$, $F(Tss_2)$, ... F(Tss); note that this is not the extension of F as above, just the normal application of a function to a set of arguments. If X_K, X_{K+1}, ... X (where $X_I \in Tss_I$) is a chain with Sup X, then $F(X_K)$, $F(X_{K+1})$, ... F(X) is a chain whose Sup is F(X). But, although $F(X_I) \in F(Tss_I)$, it is not necessarily the case that $F(X_I) \in F^*(Tss_I)$. However, if $F(X_J) \in F^*(Tss_J)$, then $F(X_{J+1}) \in F^*(Tss_{J+1})$ because $F(X_J) \in F^*(Tss_J)$ means that for all $F(Y_J) \in F^*(Tss_J)$: $F(X_J)$ is not a prefix of $F(Y_J)$ and *vice versa*, and also that: X_J is a prefix of X_{J+1} and Y_J is a prefix of Y_{J+1} imply that $F(X_J)$ is a prefix of $F(X_{J+1})$ and that $F(Y_J)$ is a prefix of $F(Y_{J+1})$. Thus $F(X_{J+1})$ is not a prefix of $F(Y_{J+1})$ nor *vice versa*, by the properties of the poset of sequences. Therefore, every chain of $F(X_I)$ (an element of $F(Tss_I)$) has a closed-above subchain $F(X_{I \geq J})$ (an element of $F^*(Tss_{I \geq J})$), so $F^*(Tss_1)$, $F^*(Tss_2)$, ... $F^*(Tss)$ is a chain and $F^*(Tss)$ is its Sup. □

The detailed proofs of the isotonicity of the determinate operators on Tagged-sequences are too long to be included here. Furthermore, the proof of isotonicity and continuity of the Arbiter requires a somewhat different approach, since it is not merely an extension of a determinate operator: in fact, it is a different recursion schema. The appendix contains the recursive definition of the Arbiter, and also a representative determinate operator, the Outbound Switch (Oswitch). The other definitions and the complete proofs are presented in [12].

Results

Since the determinate operators are isotone and continuous in the poset of data sequences, they are isotone and continuous in the poset of sets of tagged data sequences. Similarly, the Arbiter is isotone and continuous in the poset of tagged sequences. Therefore, any recursive system of equations involving these operators has a unique minimal (first order) fixed-point in that poset. This means that any DFPL program, with or without iteration (cycles in the directed graph), but without recursion, corresponds to a well defined function from sets of tagged sequences to same, and all such functions are themselves isotone and continuous.

Furthermore, since the set of continuous functions from complete posets to same, is itself a complete poset, and since composition and the first order fixed-point finding functional are

continuous in this poset, any system of recursive functional equations has a unique minimal (second order) fixed-point which is a first order continuous function. This means that DFPL programs with recursive operators correspond to well defined functions from sets of tagged sequences to same. Hence, all DFPL programs correspond to well defined functions.

Conclusions

Data flow programming languages have cleaner mathematical semantics than ordinary programing languages. This is because they are basically applicative in nature and local in effect so the functions act solely on the data without states, continuations or other complications. The tags associated with the data sequences do complicate matters of course, but this complexity is for the purpose of dealing with non-determinacy, which is not addressed by states, continuations etc. Furthermore, the tags serve double duty. First, they allow the construction of a straightforward partial order. Second, they are necessary to the specification of how operators functionally transform input sets of sequences to output ones. Hence they are less onerous than they might seem at first.

It seems reasonable to assume that the approach outlined above, namely the use of tagged sets of objects, is applicable to the mathematization of the semantics of non-determinate programs in conventional languages. However, the details remain to be worked out.

References

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

2. P.R. Kosinski, "A Data Flow Programming Language". IBM Research Report RC-4264 (March 1973).

3. D. Scott, "Outline of a Mathematical Theory of Computation". Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems, pp 169-176 (1970).

4. R.D. Tennent, "The Denotational Semantics of Programming Languages". *Communications of the ACM*, Vol. 19, No. 8 (August 1976) pp 437-ff.

5. R.W. Floyd, "Assigning Meanings to Programs". Proceedings of Symposium in Applied Mathematics, American Mathematical Society, vol. XIX (1967) pp 19-32.

6. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming". *Communications of the ACM*, Vol. 12, No. 10 (October 1969) pp 576-583.

7. G.D. Plotkin, "A Power-domain Construction". *SIAM Journal on Computing*, Vol. 5, No. 3 (September 1976).

8. M.B. Smyth, "Power Domains". Conference on Mathematical Foundations of Computer Science, Gdansk, Poland (September 1976).

9. P.R. Kosinski, "Mathematical Semantics and Data Flow Programming". ACM Third Symposium on Principles of Programming Languages (January 1976).

10. R.M. Keller, "Denotational Models With Indeterminate Operators". IFIP Working Conference on Formal Description of Programming Concepts (August 1977).

11. G. Kahn, "A Preliminary Theory for Parallel Programs". IRIA Laboratory Report 6 (January 1973).

12. P.R. Kosinski, "Denotational Semantics of Determinate and Non-Determinate Data Flow Programs". PhD Thesis, MIT Laboratory for Computer Science (in preparation).

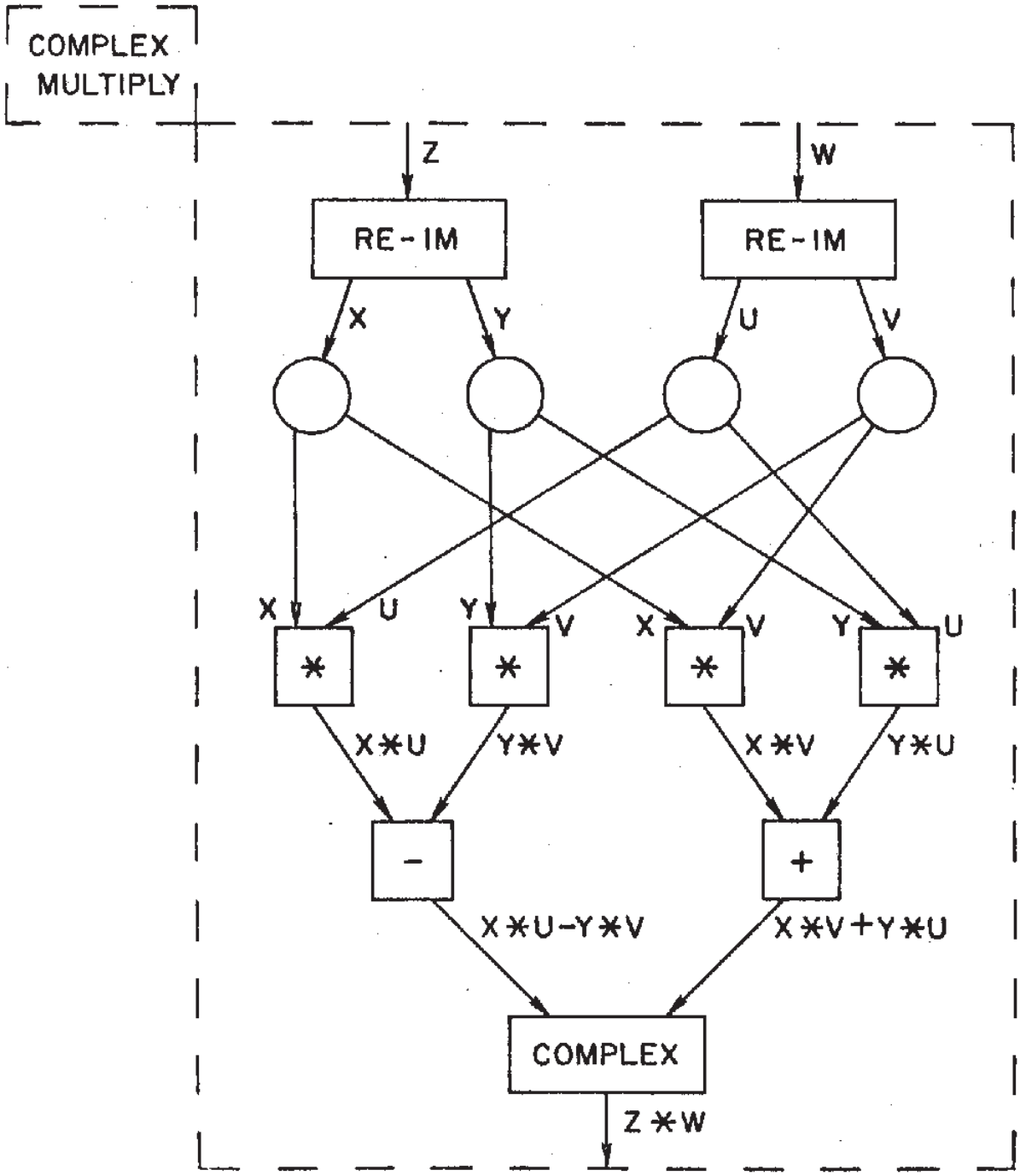
COMPLEX
MULTIPLY
Z
W
RE-IM
RE-IM
X
Y
U
V
X
U
Y
V
X
V
Y
U
*
*
*
*
X*U
Y*V
X*V
Y*U
-
+
X*U-Y*V
X*V+Y*U
COMPLEX
Z*W

FIGURE 1