MIT/LCS/TR-220

DENOTATIONAL SEMANTICS OF DETERMINATE AND NON-DETERMINATE DATA FLOW PROGRAMS

Paul Roman Kosinski

DENOTATIONAL SEMANTICS OF DETERMINATE AND NON-DETERMINATE DATA FLOW PROGRAMS

by

Paul Roman Kosinski

May 1979

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

Denotational Semantics of Determinate and Non-determinate Data Flow Programs

by

Paul Roman Kosinski

Submitted to the Department of Electrical Engineering and Computer Science
on 21 May 1979, in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

Among its other characteristics, a programming language should be conducive to writing modular programs, be able to express parallelism and non-determinate behavior, and it should have a cleanly formalizable semantics. Data flow programming languages have all these characteristics and are especially amenable to mathematization of their semantics in the denotational style of Scott and Strachey. 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 dissertation 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. Also presented are suggestions for extensions of this semantics, discussions of "fair" non-determinacy and other questions, and the relation of this approach to other approaches. In particular, it is unnecessary to use the "power domain" construction in order to handle simple non-determinacy in data flow languages.

Key Words and Phrases

Programming language semantics, data flow programming, denotational semantics, non-determinate programs, fair arbiters, parallel computation, applicative programming.

Thesis Supervisor: Professor Jack B. Dennis

To Mary, Isabel and Roman

Acknowledgements

I wish to thank the members of my thesis committee, Professors Jack Dennis, Carl Hewitt and Vaughan Pratt, for their patience and suggestions during the preparation of this thesis. I am also grateful to my colleagues Allen Brown, Vincent Kruskal and George Markowsky for their helpful discussions.

I wish to thank the IBM Fellowship Program and especially the IBM Research Division for their invaluable support.

Finally, I wish to thank my parents for their patience, and especially my wife, Mary, for her continued understanding and encouragement, without which this thesis never would have been possible.

Table of Contents

-1-
Introduction

Need for Formal Semantics

The success of syntax theory in making precise the syntax of programs led investigators to attempt to describe the semantic behavior of programs with equal precision. In particular, in order to prove theorems about the behavior of programs, it is necessary to have a mathematically precise set of axioms which define how programs behave. These axioms define the way in which the elementary semantic units behave (where elementary units are the basic data, operators, statements, etc.) and how the behavior of compound semantic constructs (such as expressions, statement lists, etc.) behave in terms of their components, elementary or compound.

The major truth one wishes to be able to prove about a program is that it does what it is supposed to do. There are two ways of expressing this: that it meets some specification, or that it does what another program does (which is known correct). Proving that one program does what another does is usually called proving program equivalence, and is not decidable in general. Proving that a program meets some specification requires having a formal statement of that specification (which also must be known correct, a point sometimes overlooked) and then proving that the behavior of the program is consistent with the specification. Such specifications (usually expressed in predicate calculus, the "assembly language" of the specification world) are often more compact than an equivalent program known to work, but they are not necessarily more perspicuous, since they may contain much that is "non-constructive" which programs by definition cannot. That a program meets its (formal) specifications is also undecidable in general.

There are other things that one might want to prove about a program. The most common is that the program terminates for all of its "legal" input, that is; one wants to prove that its domain of definition is what one thought. A second property worthy of proof is that the program consumes (no more than) a certain amount of time or space; the efficiency of a program is almost as important as its correctness. Obviously, a mathematically precise semantics for programs is needed in order to construct mathematical proofs such as these.

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, but it is rather far from the mainstream of mathematics, so that it is difficult to invoke many useful theorems or other tools. The axiomatic approach of Floyd [Flo-67] and Hoare [Hoa-69], which views a program as relating (in the mathematical sense) the "before" state of the abstract machine to its "after" state, has the disadvantages that it needs something that has a state, and that relations are less convenient than functions. The functional approach of Scott and Strachey [S&S-71] treats the semantic behavior of a program as a function from inputs to outputs, a well known kind of mathematical object.

The tractability of the formal semantics of a programming language depends more on the elegance of that language than on the class of semantic model chosen, however. We will present a programming language whose semantics we trust is quite tractable, considering its scope.

Need for Modularity

If a program is large, it is important that it be decomposable into parts, called modules, each of which performs a well defined function (at least in the informal, if not the formal, sense). Furthermore, it is important that the interactions of the modules with each other be held down to a reasonable amount. That is, the functions performed should be as independent as possible besides being well defined. The purpose of modularization, of course, is to keep the program understandable, since it is the rare person who can comprehend a large system with many interdependencies. In fact, since the number of interdependencies can grow exponentially in the number of components (consider all K way interactions, for $K \leq N$), one might consider the point of modularization to reduce such exponential growth to a more tractable polynomial or even linear growth. If the program is quite large, a hierarchy of modules is more appropriate. Then each module at the top level is composed of modules, each of which is in turn composed of modules etc., until the modules are simple enough to be understood without further decomposition [Sim-69]. This hierarchical decomposition need not be a tree, a submodules may be shared (e.g. both the carburetor and the automobile as a whole contain screws).

A program which is well partitioned into modules also is more likely to be proved correct (assuming that specifications for it can be formulated). The approach to proving such a program, as one might expect, is to prove that each module properly implements its well defined function, and that the modules are interconnected so as to meet the overall specification. If the program is a hierarchy of modules, this process is repeated for each level of the hierarchy.

Modules are often realized as subroutines, or more likely, in large programs, as collections of subroutines sharing data. An extreme case of modularization may be found in *data abstractions* as exemplified by the CLU language. Roughly speaking, data abstractions are collections of subroutines which provide and enforce access functions for an extended (e.g. user defined) data type. The way in which data abstractions differ from the ordinary way of providing access functions is that the only way to access objects of that type is through the subroutines of the data abstraction. Thus data abstractions assure that the program is modularized as claimed and that no one is "cheating" by violating the module boundaries.

This thesis presents a language which is uniquely disposed towards modularization, both in its syntax and its semantics. Since its semantics is based on the mathematical notion of function, it is possible for modules to perform one function in the formal sense. Thus this language and its semantics may provide a basis for proving properties of large programs expressed in it.

Need for Parallelism

There are several situations in which parallelism is desirable or necessary in a programming language. The first situation is when the problem to be solved is inherently parallel. The classic example of this is a multi-user computer system. Each user sits at a terminal making independent requests to the computer. Since the users are independent, and since persons live their lives in parallel with one another, it follows that the computer system must be able to serve these requests in parallel (i.e. simultaneously with respect to an appropriate time granularity). In order to serve the requests in parallel, some part of the computer's program must be capable of parallel operation. In the case of a regular time sharing system such as VM/370 (the purest case), it is only the supervisory program which operates in parallel, each user appears to have a virtual 370 on which she runs her programs sequentially. In fact, operating systems in general need parallelism [Kos-73b].

The next case in which parallelism is desirable is when there is some hardware (especially CPU) with actual parallel processing capacity and the user wishes his program to take advantage of it in order to run faster in total elapsed time. A typical example of this is performing weather (hydrodynamic) calculations on a highly parallel computer such as ILLIAC IV. Here, the extra speed gained by performing calculations simultaneously on many grid points makes the difference between useful answers and not (nobody wants to predict yesterday's weather).

These two cases of parallelism in program operation are rather different. In the first case, the simultaneous operations tend to be doing different tasks, while in the second case, they all are performing nearly the same computation, but on different data. Different programming language approaches have been adopted to cope with these different cases. To handle the first case, multi-tasking facilities often have been added to an otherwise conventional programming language. For example, PL/I has the TASK option on the CALL statement, which causes the invoked procedure to be run as an independent, parallel task or process. In ALGOL style languages, the parallel statement approach is favored; this is a compound statement whose component statements are to be executed in parallel with one another, rather than serially in the order they are written. In both these language classes, some synchronization operations are provided also because the parallel paths are never *totally* independent of one another.

The second kind of parallelism is often handled without any specialized features in the language, but is rather accommodated entirely by the compiler. For example, APL is a sequential programming language with array data and an extensive collection of array operators, but with no emphasis on parallelism. However, it is easy to imagine an interpreter or compiler for APL programs which compiles in a fashion to take full advantage of the array processing parallelism of the ILLIAC IV.

There is a third kind of parallelism that is desirable in programming that is not truly supported by any common programming language. That is parallelism for the sake of omitting unnecessary detail. The course of programming language development has been to create languages of ever "higher level", where by higher level is meant a language in which less implementation detail need be specified. For example, FORTRAN introduced the notion of arithmetic expression, LISP the notion of automatic storage management, and most recently, CLU and others have introduced the notion of abstract data type, an advanced form of data representation independence. All of these languages, however, still demand that the programmer specify that operations take place in some serial order, even though problem only demands that the operations take place in some partial order. The only case in which the exact order need not be specified is when the language has operations which operate on structured data as a whole; then the order of operation on the components need not be specified. But if one is defining an operation on structured data, the irrelevant total ordering of component operations may again creep in. For example, in defining a complex add operation, it doesn't matter whether the real parts are added first or the imaginary parts are added first.

What is needed, then, is a programming language which supports all three kinds of parallelism, parallelism demanded by the nature of the problem, parallelism demanded by the need for execution speed, and parallelism needed to suppress unnecessary implementation detail.

Need for Non-determinate Behavior

Although it is generally considered desirable for programs to be determinate (to always give the same output when presented with the same input) there are certain cases in which determinate behavior would be crippling. Consider the classic example of an airline reservation system: it consists of a central computer(s) and data-base connected to a number of agents' terminals. Each agent works independently, requesting information and booking reservations. Thus the behavior of the system must include some dependency on the arrival time of the transactions — the last seat on a flight must be given to the person who requests it first (where "first" means at least ϵ earlier). But such timing dependency is contrary to the notion of determinate behavior. If the system were to operate completely determinately, there would be no way for the system to transact with the agents at their convenience, but only according to a rigorous schedule. If one agent were out to lunch unexpectedly, for example, all the other agents would be delayed while the system waited for that agent's response. The way out of this difficulty, of course, is to include the arrival time as part of the input data, then the time dependency reduces to a data dependency, which is determinate. However, the part of the system which performs this operation is itself nondeterminate, but it is an isolated singularity.

The part of the system mentioned above which merges multiple sources of inputs into one output and perhaps tacks on the arrival time of each input is often called an *arbiter*. (The very act of merging several input streams into one output stream attaches an ordering to the arrivals of the separate inputs, so often an explicit arrival time may be dispensed with.) Given such an arbiter, which merges several streams of inputs into one output stream, a question which is of much concern is the question of fairness, that is, whether the input streams get equal treatment by the arbiter. In particular, might inputs one some ports get accepted preferentially to inputs one some other ports, or worse yet, is it possible that the inputs presented at some input port be held up indefinitely while inputs from other ports are accepted freely. Both of these behaviors are conceivable for arbiters (since, by their very name, their merging is arbitrary) but, although the priority treatment of certain inputs might be desirable, the indefinite delay of some inputs when there are no other inputs is almost certainly undesirable.

We may conclude from this discussion that a programming language must allow nondeterminacy but that it is rarely necessary to use it, and when it is, the arbiter seems to be an appropriate construct. The question then arises as to whether the nondeterminate arbiter operation which is provided is fair or not (and which meaning of fair applies). Therefore, any semantics of such a programming language surely must be able to cope with nondeterminacy and with the question of fairness.

Overview of Dissertation

In an attempt to meet the four needs outlined above, this dissertation sets forth and analyzes an unconventional kind of programming language, called a data flow programming language [Den-73, Kos-73]. This semantics of this language is defined in terms of mathematical functions, yet the functions transform the data of interest rather than the state of the machine, so modularity is achieved easily by means of function composition. The two dimensional syntax of the language provides parallelism in a natural manner, both in terms of the elimination of detail and the specification on independent tasks. The ability of the language to operate on structured data means that parallelism of such operations is also possible. Finally, the language allows non-determinate programs, and has a relatively straightforward semantics for non-determinacy, but the language construct for non-determinism allows the programmer to isolate the non-determinate behavior in small sections of the program, thus allowing the analysis of most of the program in the simpler determinate semantics.

The major part of the dissertation deals with the denotational semantics of non-determinate data flow programs. The necessary domain for the functions is defined and its properties proved; then the primitive operators in the language are functionally defined and they are proved to have the necessary mathematical properties. In particular, chapter 2 discusses and informally defines data flow programming languages, chapter 3 gives background on mathematical semantics, chapters 4 through 6 contain the formal definitions and proofs, and chapter 7 concludes with discussion of several points.

-2-
Data Flow Programming Languages

Background

In recent years a new class of programming languages, called data flow languages, has evolved [Den-73, Kos-73]. 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. A data flow program may be represented by a flowchart-like network of operators connected by data paths. Each operator executes when the data it needs is present on its input paths yielding transformed data on appropriate output paths. Operators are strictly local in effect, that is they can influence one another only by means of data sent via the paths. New operators may be defined as networks of other operators, analogous to subroutines, and recursive definitions are permitted.

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. In spite of its applicative behavior, an operator may be a function from input *sequences* to output *sequences* and thus exhibit an (internal) state with regard to single inputs and outputs. Therefore, data flow languages can be analyzed mathematically almost as easily as "toy" applicative languages (e.g. pure LISP) but are more powerful in that they provide parallelism and "state".

Informal Semantics of DFPL, a Data Flow Programming Language

The data flow language which will be considered in depth in this paper is a development of the author [Kos-73], and is called DFPL for brevity.

A DFPL program is a directed graph whose nodes are operators and whose arcs are data paths. Operators in DFPL functionally transform their inputs to their outputs without *ever* affecting the state of the rest of the program. Since there is no control flow, there is no GOTO; in spite of this, loops may be programmed as well as recursion. Most significant though, is the fact that unlike ordinary applicative languages, programs may exhibit memory behavior: that is, the current output may depend on past as well as current inputs. The effects of memory are local like those of other operators and it does not permeate the semantics of programs.

Data in DFPL are pure values, either simple like numbers or compound like arrays and records. There are no addresses as primitive data in DFPL, although compound operators may be defined to interpret data values in a manner reminiscent of addresses. An operator "fires" when its required inputs are available on its incoming paths. After an unspecified interval, its 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. A synchronous operator is one which fires only when all its inputs are present and it produces all its outputs at once. The outputs may depend on past inputs as well as current inputs. If the outputs of a synchronous operator depends only on current inputs, the operator is said to be simple. Synchronous operators are analogous to subroutines (with "own" or "static" variables if the operator is not simple). Some operators produce a time sequence of output values from one input value or conversely; they are analogous to coroutines. The operators in a DFPL program thus operate in parallel with one another subject only to the availability of data on the paths.

An operator may either be primitive or defined. An operator is defined as a network of other operators connected by data paths such that some paths are connected at one end only. These paths are the parameters of the defined operator. An instance of a defined operator operates as if its node were replaced by a copy of the network which defines it and the parameter paths spliced to the paths which were connected to that node. This "copy rule" allows recursive operators to be defined.

Sufficient synchronization signals are passed with the data on the paths so that operators do not fire prematurely, and so that operation of the program as a whole is independent of the timings of the component operators (at least in basic DFPL, full DFPL allows timing dependent programs in order to cope with the real world).

Classes of Operators

There are three classes of operators in DFPL: simple operators, including the usual arithmetic, logical and aggregate operators 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 streams to streams 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 streams to streams, or more profitably, as we shall soon see, as functions from sets of streams to sets of streams.

Synchronous operators allows us to avoid the tedium of using a separate index for the stream of values on each data path. All paths in a subnetwork of synchronous operators may share the same stream index since that subnetwork behaves as a single synchronous operator. Note that all simple operators are synchronous and stream operators may or may not be synchronous. Also, any defined operator constructed entirely out of synchronous operators is itself synchronous.

Primitive Operators

There are five primitive operators in basic DFPL (shown in Figure 2.1). Of these, two are simple in their behavior: the *Fork* and the primitive computational function or *Pcf*. The *Fork* is a multi-output identity function, that is, a copy of its input is sent to each of its outputs. The *Pcf* is really a whole set of operators including the usual arithmetic, logical and aggregate operators (e.g. *Construct* and *Select*). The *Modify* operator is an example of a *Pcf* which typifies DFPL in that it generates new data rather than updating existing data. *Modify* takes three inputs, an array A, an index I and a value V, and produces one output, a new array *Anew*, which is a copy of A except that $Anew_I = V$. Note that the *Fork* and all *Pcf* operators are synchronous. Since *Forks* have such simple functional properties we do not treat them as explicit operators on proofs, but rather just label all their paths the same.

The most complicated of the primitive operators are the *Switch* operators, also shown in Figure 2.1. These two operators have the property that each firing is independent of previous firings, but not all inputs/outputs are demanded/produced upon each firing. The outbound *Switch* or *Oswitch*, for example, demands C and U as inputs for each firing, but only one of X, Y and Z receives output in any firing. Which one receives the output, which is just the input value U, is determined by the value of the input C. The inbound *Switch* or *Iswitch* operates conversely, only one of the inputs X, Y, Z is accepted upon firing (C is always demanded), and its value is always sent out on U.

Informally speaking, an *Iswitch* merges two or more data streams into one data stream of the same length as the control stream, selecting which input data stream to use next according to the current value on the control stream. Conversely, an *Oswitch* splits a data stream into two or more data streams dependent on the values of the control stream. Figure 2.2 exemplifies the behavior of both *Iswitch* and *Oswitch* according to which paths are input and which are output. In both cases, the ordering(s) of the output stream(s) is consistent with the ordering(s) of the input stream(s). Although the value of an output from a *Switch* is dependent only on the current input values for this firing, the position of that output value in its stream is dependent on previous firings, hence neither *Switch* is a simple operator.

Since these operators sometimes do not demand/produce inputs/outputs, 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 of inputs to streams of outputs.

The most interesting primitive operator in basic DFPL is that which behaves like a kind of memory cell. It is just a holding station, that is, the output is what the input was on the previous firing and the initial output is its constant parameter. That is, $Out^{J+1} = In^J$ and $Out^1 = Q$. The *Hold* operator is interesting because it is sufficient to construct any kind of memory desired, yet itself is purely functional (albeit from input streams to output streams). It can also be used to construct iteration.

All of the above primitive operators are causal in the sense that an output cannot be affected by future inputs; that is, once an output is produced, it cannot be changed.

Some Compound Operators

Switch operators are most often used in matched pairs, with the control input of each connected, via a *Fork*, to the same source of a control stream. When connected in this way, the DFPL version of a conditional expression results, as shown in Figure 2.3. The equivalent expression is ***If*** $P(X)$ ***Then*** $F(X)$ ***Otherwise*** $G(X)$.

Figure 2.4 shows a definition of a repeating constant operator. This operator takes no inputs but produces an (infinite) stream of output values, all the same (Q).

A fancier memory cell is shown in Figure 2.5. When a 0 value is presented on the control path C, the current contents is read out onto path Y. When a 1 value is presented on C, and a data value presented on input path X, the cell is updated to contain that new data value. The cell has an initial contents of Q.

The Primitive Non-determinate Operator

To allow the construction of programs with indeterminate behavior, we define an operator which merges its input streams in an arbitrary manner. This operator, called the *Arbiter*, is shown in Figure 2.6. Speaking informally, the *Arbiter* operator merges two or more input streams into an output stream whose order of items is consistent with the separate orders of items in the input streams. This merging is done randomly (or arbitrarily) analogous to shuffling together two decks of cards. The *Arbiter* also (optionally) generates a stream of control values which tells exactly how the merge was performed. This control stream is of a form such that if it is fed to an *Oswitch*, the merged stream can be unscrambled into its component input streams. The optional form of *Arbiter* can be programmed from the more primitive form, which does not generate the control stream output, together with a *Fork* and an *Iswitch*.

Since the *Arbiter* produces an output stream chosen randomly from a set of possible output streams, we might characterize the *Arbiter* as a relation from input streams to output streams. However, since the the fixed-point theory of functions is better understood, we will treat the *Arbiter* as a function from (sets of) input streams to sets of output streams. We consider sets of input streams even though intuitively the *Arbiter* works on individual input streams because we wish the domains and codomains to be compatible.

In extending the semantics of DFPL to accommodate the *Arbiter*, the semantics of the determinate operators must be upgraded also. This upgrading is the obvious one of saying that the determinate operators map sets of input streams into sets of output streams pointwise, that is, each stream in the input set gets mapped to a single stream in the output set by applying the old stream-to-stream function of the operator. Multiple input operators are more complicated. If the input sets originate at the same *Arbiter*, then the operator is applied to corresponding streams from the input sets in a manner similar to an inner product of vectors. If the sets originate at different *Arbiters*, then the operator is applied to the Cartesian product of the sets. If the sets have mixed origins, that is have some *Arbiter* in common which affected their computation, as well as independent *Arbiters*, then a mixture of inner and outer (Cartesian) products must be taken. Thus, the determinate operators produce output sets whose cardinalities are no bigger than the product of the cardinalities of the input sets. The indeterminate *Arbiter*, unfortunately, tends to cause cardinalities to get out of hand, since the output set cardinalities depend on the input set elements (i.e. the number of ways they can be merged) as well as the input sets cardinalities.

Examples

The DFPL program shown in Figure 2.7 is an example of a procedure definition. The procedure performs the multiplication of two complex numbers with a high degree of parallelism. Figure 2.8 shows a DFPL procedure for computing the mythic recursive factorial function. Figure 2.9 shows a DFPL procedure which implements a random access memory of 1000 cells, each initialized to 0.

The program illustrated in Figure 2.10 takes advantage of the fact that the optional control output of an *Arbiter* may be used to control an *Oswitch* to unscramble the merging performed by that *Arbiter*. If the operator F is simple, that is, it is a function from its current input to its current output (thus independent of previous inputs), then the defined operator *Triple-f* behaves exactly as three copies of F applied separately to U, V, and W producing X, Y, and Z respectively (see Figure 2.11). However, the operator F is shared among the three input and output paths and therefore saves resources as compared to three copies of F. Of course, this is at the cost of running at least three times slower. Most important, even though the internals of *Triple-f* are indeterminate, the behavior of *Triple-f* as a whole is functional and thus determinate. Therefore, it is possible to construct determinate programs using indeterminate components, and furthermore, proving one has done so is not necessarily difficult.

Other Data Flow Languages

One of the earliest pure data flow models of programming was developed by Rodriguez [Rod-67] This provided most of the capabilities of DFPL except for operator definition, and, thus, recursion. Programs in this language were guaranteed determinate in operation.

Luconi developed a model of parallel computation [Luc-68] which was more general in some ways than Rodriguez's. However, because a relatively conventional sort of memory cell was necessary to hold data for the operators (approximately one such cell per operator), determinate behavior could not be guaranteed, except by following strict conventions in programming.

Adams developed a pure data flow programming language [Ada-68] similar to Rodriguez's and DFPL except that data paths were FIFO queues of unbounded length. This makes direct hardware implementation impossible; it is possible for DFPL without recursion if data types are of bounded size (e.g. FORTRAN numbers and arrays). It is presumably possible to directly implement Rodriguez's language in hardware also.

At the same time as DFPL was developed, Dennis independently developed a Data Flow Procedure Language [Den-73] which is almost identical in terms of its primitive concepts. In its original form, it lacked an indeterminate primitive operator (present in DFPL) so that indeterminate programs could not be constructed. Further restrictions on the construction of programs in Dennis' language were imposed to ease the mathematization of the semantics. These restrictions also simplify direct execution by a data flow processor (hardware). Thus, certain semantic behaviors, permissible in our DFPL, were not allowed in Dennis' original language.

Of the four languages mentioned here (other than DFPL), only Dennis' is having a denotational semantics developed for it. Stoy [Sto-74] and Ciccarelli [Cic-76] have mathematized the semantics of this language.

A related class of programming languages is those conventional languages which include interprocess communication mechanisms. Examples of these are suggested by Hoare's "communicating sequential processes" [Hoa-78] and Kahn and MacQueen's "coroutines and networks of parallel processes" [K&M-78]. Yet another kind of language related to data flow languages is LUCID of Ashcroft and Wadge [A&W-77]. This is a language which is applicative yet works on streams of data.

-3-
Background on Mathematical Semantics

Kinds of Mathematical Semantics

The two approaches to mathematical semantics are, as stated earlier, the axiomatic and the denotational or functional. In the axiomatic approach, each primitive operation in the programming language has associated with it one or more axioms which formally specify the effect the operation has on the state of the abstract machine when that operation is executed. That is, the axioms describe the mathematical relationship between the "before" state and the "after" state. This relationship may or may not be functional. A sequence of operations have an effect which is the composition of the individual relations for the component operations. A loop in the program requires an inductive proof based on the relationship implied by the loop body and loop predicate. The inductive assumption (often called the loop invariant) may either be given or deduced from the initial and final conditions. A recursive program requires an inductive argument also.

Programs which modify data structures as side effects are hard to deal with in any semantics. The usual axioms for assignment are not directly applicable when the assignment is to some computed variable. This situation arises with assignment to array components, with assignments indirectly via pointers, and with "aliasing" of any data objects via procedure parameters. This remains one of the open problems in axiomatic semantics [C&O-78].

As is well known, programs which loop or recur sometimes do not terminate. Unfortunately, the inductive proof of a loop's behavior mentioned above often does not prove termination, but only the behavior of the loop if it terminates. The termination property (often) must be proved as a separate result. A new axiomatic semantics called *dynamic logic*, which is based on modal logic [H&P-78], allows one to treat termination simultaneously with "partial correctness", (as the behavior assuming termination is frequently called). An extension to dynamic logic allows one to treat non-determinacy as well [H&P-78].

Thus, given a program together with a set of assertions about its behavior, one might determine by theorem proving (manual or automatic) whether the program satisfies its assertions. Alternatively, one might derive an assertion which describes the program's behavior.

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 (in the axiomatic approach, an inductive proof is needed). Setting up the functional equation corresponding to the loop, one gets:

$$F(X) = \textit{If } Test(X) \textit{ Then } F(Body(X)) \textit{ Otherwise } X$$

where *Test* is the predicate of the WHILE, *Body* is the function which describes the body of the loop, and *F* is the function which describes the loop as a whole. This is a recursive definition, but it is hard to solve because the unknown, *F*, is a *function*.

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

Mathematical Concepts

A *partially ordered set*, or *poset*, is a set of objects together with a relation which is reflexive, transitive and antisymmetric. That is, $\forall X: X \leq X$, $\forall X, Y, Z: X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z$, and $\forall X, Y: X \leq Y \wedge Y \leq X \Rightarrow X = Y$. If, in addition, the relation holds in one direction or the other for every pair of elements in the set, the set is said to be *totally ordered*. The integers are totally ordered under the usual ordering, while the set of all subsets of a given set are partially ordered under the inclusion relation. A *chain* is a totally ordered (subset of a) poset. A set is said to be *pre-ordered* or *quasi-ordered* under a relation if the relation is reflexive and transitive (but not antisymmetric). The set of equivalence classes under the quasi-order form a partially ordered set, where X and Y are in the same equivalence class iff $X \leq Y \wedge Y \leq X$.

A Cartesian product of posets is itself a poset under the pointwise partial order; that is, $\langle Xa, Ya, Za\rangle \leq \langle Xb, Yb, Zb\rangle$ iff $Xa \leq Xb \wedge Ya \leq Yb \wedge Za \leq Zb$. Since a function can be treated as an element of a large cartesian product (the product of identical copies of the codomain indexed by the domain), functions can be partially ordered also. The order is defined by: $F \leq G$ iff $\forall X: F(X) \leq G(X)$

An *upper bound* of a subset S of a poset P is an element $U \in P$ such that $\forall X \in S: X \leq U$. A *supremum* or *least upper bound* (often abbreviated *l.u.b.*) of a subset S is an element $L \in P$ such that $\forall U \in P: L \leq U$ where the U's are upper bounds of S. An *infimum* or *greatest lower bound* (often abbreviated *g.l.b.*) is the *order duals* of the above (which is obtained by replacing "$\leq$" by its converse relation "$\geq$"). Many posets of interest have a least element, called *bottom* ("$\bot$"), which forms a lower bound for all subsets. A *lattice* is a poset in which every two elements have both an infimum or *meet* and a supremum or *join* [M&B-67]. Note that M is the meet of X and Y iff $M \leq X$ and $M \leq Y$ and $\forall B: B \leq X \wedge B \leq Y \Rightarrow B \leq M$, and the join is the order dual. Many lattices of interest have both a least and a greatest element.

A function from a poset to a poset is said to be *isotone* iff $\forall X, Y: X \leq Y \Rightarrow F(X) \leq F(Y)$. (The term *monotone* is often used instead of isotone, but it is less precise since isotone corresponds to monotone *increasing* only [Ros-77].)

A poset is said to be *chain-complete* iff every chain has a supremum (not necessarily in the chain itself). The integers are not chain complete, for example, but the real numbers on a closed interval are. Any chain complete poset necessarily has a bottom element — it is the supremum of the empty chain. A more interesting example of a chain complete poset is the set of all finite and infinite sequences of elements of some set partially ordered by the *prefix* or *initial subsequence* relation. (For example, $AB \leq ABC \leq ABCD$ and $AB \leq ABD$.) In this poset, the suprema are the infinite sequences (length ω). The empty sequence is the least element of the entire poset, but there is no greatest element. The poset may be pictured as a tree of infinite depth, where each (finite) node corresponds to a (finite) sequence, and at each node, each arc leading away from the node is labeled with a different element from the underlying set of objects. We restrict our attention to chains which are countable, which does not require the underlying set to be countable. For example, the set of subsets of the integers is uncountable, but the chains under the inclusion order each have a countable number of elements. Although there are other varieties of completeness, such as *directed-set completeness* in which any finite subset which has an upper bound has a supremum, from now on we will mean "countable chain complete" whenever we say "complete".

A function from one complete poset to another is said to be *continuous* iff it is isotone and preserves suprema; that is, iff the value of the function on the supremum of a chain in its domain is the supremum of the set of values which is the image of that chain. Note that since the function is isotone, it maps chains into chains. Also, note that the isotonicity of the function can be deduced from its continuity (suprema preservation), merely by considering finite chains, whose suprema are their greatest elements. It is an easily proved and useful fact that a function which maps a cartesian product of (complete) posets into a poset and is isotone (continuous) on each argument is also isotone (continuous) on the tuple. Similarly, the composition of two isotone (continuous) functions is in turn isotone (continuous). It is also straightforward to prove that the set of continuous functions from one complete poset to another itself forms a complete poset under the natural partial order on functions defined above.

Now we come to the point of introducing posets, completeness, isotonicity and continuity — the Tarski fixpoint theorem [Mar-76]. If $F: P \rightarrow P$ is an isotone function mapping a complete poset into itself, then F has a least fixpoint X. That is $\exists X: F(X) = X$ and $\forall Y: F(Y) = Y \Rightarrow X \leq Y$. Furthermore, if F is continuous as well as isotone, we can "compute" its least fixpoint by a straightforward technique (actually the technique involves taking a limit, so it is not computable in the ordinary sense) [Sto-77]. Consider the sequence $\perp$, $F(\perp)$, $F^2(\perp)$, $F^3(\perp)$, $F^4(\perp)$, etc. This sequence forms a chain because $\perp \leq F(\perp)$ by definition of $\perp$ and $F^I(\perp) \leq F^{I+1}(\perp)$ because F is isotone. Therefore the sequence forms a chain which has a limit or supremum because the poset is chain complete. If we form $\bigcup_I F^I(\perp)$ (where $I \in \omega$), and call it X, we see that X equals $\bigcup_I F(F^I(\perp))$ which by continuity equals $F(\bigcup F^I(\perp), I)$ which equals $F(X)$ a fixpoint of F. To show X is the least fixpoint, assume Y is a fixpoint. Then $\perp \leq Y$ and thus $F(\perp) \leq F(Y) = Y$. So by induction, $F^I(\perp) \leq Y$ and so $X \leq Y$.

Denotational Semantics

In the standard treatments of denotational semantics one data element or function is said to be less than another iff it is less well defined than the other, so that the partial order is an ordering by approximation or information content [S&S-71]. For example, in the case of partial functions one may be said to be greater than another if it *extends* the other, that is, it is defined on a larger domain and they agree in value on the smaller domain [Man-74]. In the case of simple data, a "flat" partial order is often used, it consists of a set of data which are not order related to each other and a bottom element which is less than any true datum. Such flat posets are not of interest in themselves, but are used to construct more interesting posets as outlined below.

Earlier we indicated that it was difficult to assign an overall functional behavior to programs with loops because a recursive equation results which has a function as the unknown. Such equations can be solved in certain circumstances by means of the Y, or fixed point, operator. This is complicated by the fact that in many programming languages, such as LISP, BCPL and even PL/I, we wish to be able to treat functions as data objects. In order to mathematize this, we require that the domain of functions include functions from that domain to that domain. This means that the domain must be recursively defined. If we let N be the domain of non-functional data (such as numbers), then the domain D must be isomorphic to $N \oplus [D \rightarrow D]$, the disjoint union of N and the set of (continuous) functions from D to D. Scott's contribution has been to show that there exist lattices called reflexive domains which satisfy this isomorphism and in which the Y operator can always apply to give the unique minimal fixed point solution of the functional equations alluded to earlier. Furthermore, such domains adequately characterize programming languages. Thorough treatments of this approach to programming language semantics may be found in [M&S-76, Sto-77].

A slightly more recent approach to denotational semantics uses complete posets rather than complete lattices [ADJ-77, Mar-76, Ros-77]. Such posets accurately model the basic notion of approximation, and although the bottom element corresponds to "undefinedness", the "top" element of the lattice (and other joins necessary for the lattice to exist) do not seem to correspond to any computationally meaningful object. We do not use reflexive domains in this thesis, as we do not allow function valued data, but we do use posets rather than lattices, for the reasons stated.

Notation

Names of variables and functions are denoted by capitalized cursive italic words such as *Var* and *Fun*. This is more like programming notation than conventional mathematical notation, which tends to use single character symbols for all variables and functions, but it is more mnemonic and thus more readable when many names exist. Literal data symbols are represented by austere italic letters such as A and by austere numbers such as 999. Literal data symbols may be *tagged* by appending strings of miniature digits to the symbols, for example, $\langle A_{0}, A_{00}, A_{001}\rangle$.

The angle brackets "$\langle$" and "$\rangle$" are used to enclose explicit sequences of data, for example $\langle A , B , C\rangle$. Braces (*i.e.* "{" and "}") denote sets in the usual way: $\{X, Y\}$ denotes the set consisting of X and Y, while $\{X \mid P(X)\}$ denotes the set of all X satisfying $P(X)$.

Subscripts on names denote selection of a particular item from a set of similar items, for example Var_2, Fun_I. Superscripts on variables which are *streams* (sequences) denote selection of an element from that sequence, for example, S_N^I denotes the I-th element of the stream S_N, which in turn is the N-th stream of a set of related streams. Superscripts on data symbols mean repetition of that symbol, for example $\langle A^K\rangle$ denotes a sequence of K A's. Superscripts on function names either denote repeated composition, if the superscripts are numeric constants or variables, or they denote a new function related to that function denoted by the un-superscripted name, if the superscript is a greek letter. For example, $F^M(X)$ denotes the M fold application of F to X, whereas F^ξ denotes the "extension" of F by some rule, and F^ω denotes the "completion" of F by some other rule.

Finally, conventional mathematical notation is used for everything else: infix operators, prefix operators, quantifiers (with ":" separating the quantification from the body), function application and argument lists, and conditional expressions, including "***If***", "***Then***" and "***Otherwise***".

-4-
Semantics of Determinate DFPL Programs

Overview

In this chapter we develop the fixed-point semantics of the determinate subset (really a sub-algebra) of DFPL. To do this we first show that the domain of *Streams* is suitable for fixed-point solutions of programs, then we show that the determinate operators are continuous on this domain. We therefore deduce that (recursion free) determinate DFPL programs have a well defined behavior no matter what inputs they receive. We conclude with an example of a simple fixed-point computation of a program containing a loop.

A Complete Partial Order on Streams

A *Datum* is an element of some set of data, for example, integers, characters, Booleans, arrays of floating point numbers, payroll records, directed graphs etc. The data sets available depend on the kind of DFPL programs being analyzed. We will not consider what types of data are available except that we shall assume that the integers are since they are needed to control the Switch operators. All data are assumed to be incomparable from the denotational point of view. That is, any ordering of data in a data set (e.g. the integers) is not of interest to us since it does not represent approximation.

A stream is a finite, empty or infinite sequence of data items, often denoted by enclosing their elements in angle brackets, for example, $\langle\,\rangle$ for the empty stream, $\langle A , B , C , D\rangle$ for a finite stream of length 4, and $\langle A , B , \ldots , Z , \ldots\rangle$ for an infinite stream. More precisely, a stream is a function from the positive integers or some initial segment thereof (including the empty set), to the set of *Data*. That is, $S\colon Nseg \rightarrow Data$; where $Nseg = \{\,\}$ (S the empty stream), or $Nseg = \{I \mid 1 \leq I \leq N\}$ (S a finite stream), or $Nseg = \{I \mid I \geq 1\}$ (S an infinite stream). Put another way, streams are functions whose domains are ordinals no bigger than ω and whose codomains are some set *Data*. We denote the value of a stream at some integer I by S^I, using superscripting for emphasis. A stream S_1 is said to be a "prefix" of a stream S_2 (denoted $S_1 \leq S_2$) iff $Dom(S_1) \subseteq Dom(S_2)$ and S_2

restricted to $Dom(S_1)$ (denoted $S_2 \upharpoonright Dom(S_1)$) is equal to S_1. That is, $S_1^I = S_2^I$ for all $I \in Dom(S_1)$.

Theorem 4.1: The prefix relation is a partial order on streams.

Reflexivity and transitivity follow from the reflexivity and transitivity of "$\subseteq$" and especially "$=$". The antisymmetry of "$\leq$" follows from the antisymmetry of "$\subseteq$". Hence the set of streams form a discrete poset which we call *Cpo-streams*. $\boxtimes$

Note that the bottom element of this poset is the empty stream (denoted $\perp$ or $\langle\,\rangle$), and the infinite streams are maximal elements of the poset.

Lemma 4.1: An indispensable property of *Cpo-streams* is the following: if S_1, S_2 and S_3 are streams such that $S_1 \leq S_3$ and $S_2 \leq S_3$, then either $S_1 \leq S_2$ or $S_2 \leq S_1$. This essentially says that the graph of the partial order is a tree, with the empty stream as the root and the infinite streams as the leaves.

This follows easily from the fact that $Dom(S_1)$ and $Dom(S_2)$ are ordinals so $Dom(S_1) \subseteq Dom(S_2)$ or $Dom(S_2) \subseteq Dom(S_1)$. We apply the definition of "$\leq$" to get $S_3 \upharpoonright Dom(S_2) = S_2$, and $S_3 \upharpoonright Dom(S_1) = S_1$, Now, assuming $Dom(S_1) \subseteq Dom(S_2)$, we get $S_2 \upharpoonright Dom(S_1) = S_3 \upharpoonright Dom(S_2) \upharpoonright Dom(S_1)$ which implies $S_3 \upharpoonright Dom(S_1) = S_1$, which means $S_1 \leq S_2$. Assuming $Dom(S_2) \subseteq Dom(S_1)$ gives us $S_2 \leq S_1$, which proves the property. $\boxtimes$

Lemma 4.2: If S_1 and S_2 are infinite, $S_1 \leq S_2$ iff $S_1 = S_2$.

Theorem 4.2: The poset *Cpo-streams* is countable chain complete.

To show that this poset is chain complete, we must prove that any chain of streams has a supremum in the poset. The chains are just sets of streams, $\{S_1, S_2, \ldots\}$, such that $S_1 < S_2 < \ldots$; we need not worry that $S_I = S_J$ since a chain is a set. There are four cases to be considered: if the chain is empty, then its supremum is $\perp$. If the chain is finite, then its supremum is just its maximal element. If the chain is infinite and contains an infinite stream S, then S is the supremum of the chain, since $S \leq S$ and for all finite S_I, $S_I < S$, and for no finite S_N is $S_N \geq S_I$ for all finite S_I, and no other infinite stream can be in the chain. If the chain is infinite but contains only finite streams, then its supremum S is not in the chain but does exist in the poset. We merely define S to be the stream such that $S^I = S_N^I$ for all $I \in Dom(S_N)$ for any S_N in the chain. S is well defined because the

S_N are elements of a chain. S is infinite because the chain is infinite and the domains $Dom(S_N)$ are unbounded. No element of the chain is an upper bound (given any stream in the chain, we can find a longer one) so S is the supremum. $\boxtimes$

Therefore *Cpo-streams* lives up to its name: it is a chain complete partially ordered set. Note that the finite streams in this poset constitute a *basis* in the sense of [M&R-76]. Strictly speaking, we should define *Cpo-streams(Data-type)* and therefore have different posets for each kind of data. We will not do this in this dissertation, as the generalization is clear. Instead we will treat DFPL as an untyped language (like LISP).

DFPL Operators as Isotone Functions on Finite Streams

In this section, we restrict our attention to DFPL operators on finite streams because we wish to prove isotonicity — continuity is treated later.

The simple operators, since they operate on streams element by element, are clearly isotone. Let Sop_F be a simple N-ary operator on streams which applies F to each N-tuple of corresponding input data. We denote this by $Sop_F(S_1,\ldots,S_N)$: each function F gives a different simple operator. That is:

$$Sop_F(S_1,\ldots,S_N)=S$$

$$\textbf{\textit{Where}}\ S^I=F(S_1{}^I,\ldots,S_N{}^I)$$

$$\forall I\in L=\textstyle\bigcap_{J\leq N} Dom(S_J)$$

Now let $S_K \leq Sx_K$, then $Sx_K=\langle\,\rangle$ implies $S_K=\langle\,\rangle$ (because $Dom(S_K)\subseteq Dom(Sx_K)$). But $Sx=Sop_F(S_1,\ldots,Sx_K,\ldots,S_N)$ iff for all $I\in Lx=Dom(Sx_K,\bigcap_{J\neq K}Dom(S_J))$ $Sx^I=F(S_1{}^I,\ldots,Sx_K{}^I,\ldots,S_N{}^I)$. Now since $L\subseteq Lx$, it is also the case that for all $I\in L$: $Sx^I=F(S_1{}^I,\ldots,Sx_K{}^I,\ldots,S_N{}^I)$. But since $L\subseteq Dom(S_K)$ and $S_K\leq Sx_K$, we have $Sx^I=F(S_1{}^I,\ldots,S_N{}^I)$ for all $I\in L$. Thus $Sx^I=S^I$ for all $I\in L$ so $S\leq Sx$. Therefore *Sop* is isotone in each argument. $\boxtimes$

The operator *Hold* takes a constant datum C and attaches it to the front of the stream S. We denote this by $Hold_C(S)$: each value of C gives rise to a different *Hold* function.

We define $Hold_C(S) = C \oplus S$ where "$\oplus$" is defined as follows (with "+" representing ordinal addition):

$$S \oplus A = Sa$$

$$\textit{Where } Dom(Sa) = 1 + Dom(S)$$

$$\textit{And } Sa^I = \textit{If } I = 1 \textit{ Then } A \textit{ Otherwise } S^{I-1}$$

Thus "$\oplus$" is isotone since if $S_1 \leq S_2$ then $A \oplus S_1 \leq A \oplus S_2$. Therefore $Hold_C(S)$ is isotone in S. ⊠

We define another useful isotone operation "τ" as follows (with "$-$" representing ordinal subtraction):

$$\tau\, S = Sd$$

$$\textit{Where } Dom(Sd) = Dom(S) - 1$$

$$\textit{And } Sd^I = S^{I+1}$$

Obviously, "τ" is isotone since if $S_1 \leq S_2$ then $\tau S_1 \leq \tau S_2$. The "$\oplus$" and "τ" operations are equivalent to CONS and CDR in LISP.

The Switch operators are the most complicated functions from streams to streams. First we define the Outbound Switch operator $Oswitch_p(C,D)$:

```
$Oswitch_p(C,D) =$
    $\textit{If } C = (\,) \vee D = (\,) \textit{ Then } (\,)$
    $\textit{If } C^1 = P \textit{ Then } D^1 \oplus Oswitch_p(\tau\, C, \tau\, D)$
    $\textit{Otherwise } Oswitch_p(\tau\, C, \tau\, D)$
```

Here P is the port number, C is the control stream and D is the data stream being switched. Thus an Outbound Switch with three ports (0, 1 and 2) would require the three functions $Oswitch_0(C,D)$, $Oswitch_1(C,D)$ and $Oswitch_2(C,D)$ for its complete description.

We prove that $Oswitch$ is isotone in the argument C by showing that if $C \leq Cx$ then $Oswitch_p(C,D) \leq Oswitch_p(Cx,D)$. The proof proceeds by induction on the finite ordinal $Dom(Cx)$. Note that $Cx = (\,)$ iff $Dom(Cx) = \{\,\}$ and that $Oswitch_p((\,),D) = Oswitch_p(C,(\,))$

$= \langle\ \rangle$ Substituting Cx in the definition of *Oswitch*, we get:

$$\begin{aligned}&Oswitch_p(Cx,D) =\\ &\quad \textbf{\textit{If}}\ Cx=\langle\ \rangle \vee D=\langle\ \rangle\ \textbf{\textit{Then}}\ \langle\ \rangle\\ &\quad \textbf{\textit{If}}\ Cx^1=P\ \textbf{\textit{Then}}\ D^1 \oplus Oswitch_p(\tau\, Cx, \tau\, D)\\ &\quad \textbf{\textit{Otherwise}}\ Oswitch_p(\tau\, Cx, \tau\, D)\end{aligned}$$

We assume in the steps that follow that D is not $\langle\ \rangle$, since for any C and Cx, $Oswitch_p(C, \langle\ \rangle) = \langle\ \rangle = Oswitch_p(Cx, \langle\ \rangle)$. The base step is as follows: $Cx = \langle\ \rangle$ implies $C = \langle\ \rangle$ so that $Oswitch_p(C,D) = Oswitch_p(\langle\ \rangle, D) = Oswitch_p(Cx, D)$. The induction step is: let $Dom(Cx) = N+1$; if $C = \langle\ \rangle$, then $Oswitch_p(C,D) = \langle\ \rangle$ which is a prefix of any stream. If $C \neq \langle\ \rangle$ then $\tau C \leq \tau Cx$ and $C^1 = Cx^1$. Now if $C^1 = P$ then:

$$Oswitch_p(C,D) = C^1 \oplus Oswitch_p(\tau\, C, \tau\, D)$$

$$Oswitch_p(Cx,D) = Cx^1 \oplus Oswitch_p(\tau\, Cx, \tau\, D)$$

So $Oswitch_p(Cx,D) = C^1 \oplus Oswitch_p(\tau\, Cx, \tau\, D)$

By the isotonicity of "$\oplus$", $Oswitch_p(C,D) \leq Oswitch_p(Cx,D)$ if $Oswitch_p(\tau C, \tau D) \leq Oswitch_p(\tau Cx, \tau D)$ But $\tau C \leq \tau Cx$ and $Dom(\tau Cx) = N$, so we may assume, as the inductive hypothesis, that $Oswitch_p(\tau C, \tau D) \leq Oswitch_p(\tau Cx, \tau D)$. Now if $C^1 \neq P$ then:

$$Oswitch_p(C,D) = Oswitch_p(\tau\, C, \tau\, D)$$

$$Oswitch_p(Cx,D) = Oswitch_p(\tau\, Cx, \tau\, D)$$

But $\tau C \leq \tau Cx$ and $Dom(\tau Cx) = N$, so again we apply the inductive hypothesis, that $Oswitch_p(\tau C, \tau D) \leq Oswitch_p(\tau Cx, \tau D)$. ⊠

The proof that $Oswitch_p(C,D)$ is isotone in D is essentially identical.

The Inbound Switch operator has $N+1$ data ports D_0 through D_N, where we start at

0 because the simple case of D_0 and D_1 corresponds naturally to a ***True/False*** Switch. *Iswitch* is defined as follows:

```
$Iswitch(C, D_0, \ldots, D_N) =$
  $If\ C = \langle\,\rangle\ Then\ \langle\,\rangle$
  $If\ C^1 = 0 \wedge D_0 \neq \langle\,\rangle\ Then$
    $D_0{}^1 \oplus Iswitch(\tau\, C, \tau\, D_0, \ldots, D_N)$
  •    •    •    •    •
  $If\ C^1 = N \wedge D_N \neq \langle\,\rangle\ Then$
    $D_N{}^1 \oplus Iswitch(\tau\, C, D_0, \ldots, \tau\, D_N)$
  $Otherwise\ \langle\,\rangle$
```

One can prove that *Iswitch* is isotone in each of its arguments in a manner similar to that *Oswitch* by which was proved isotone, except that the induction must be on the (ordinal) sum of the domains of the D_I since only one is reduced by each recursion.

DFPL Operators as Continuous Functions on *Cpo-streams*

Having defined all the primitive determinate DFPL operators as functions on finite streams, we wish to complete them to be continuous functions on *Cpo-streams*. That is, we wish $\bigsqcup Op(C) = Op(\bigsqcup C)$ for any chain C (where $\bigsqcup S$ denotes the supremum of S). This is straightforward since we have yet to say how a primitive operator transforms an infinite stream. Let $Max\text{-}chain(S) = \{S_I \mid S_I \leq S\}$ for any S, so that for infinite S, $Max\text{-}chain(S)$ is the (infinite) maximal chain containing S. To make an operator continuous, we define $Op^\omega(S) = Op(S)$ when S is a finite stream, and $Op^\omega(S) = \bigsqcup Op(Max\text{-}chain(S) - \{S\})$ when S is an infinite stream (recall that if X is a set, $F(X) = \{F(Xelt) \mid Xelt \in X\}$).

Theorem 4.3: The completion Op^ω of Op as defined above is continuous.

Since *Cpo-streams* is chain complete, the supremum exists. Since $S = \bigsqcup Max\text{-}chain(S)$ we have continuity on maximal chains automatically. Now consider $\bigsqcup Op(C)$ where C is an arbitrary chain. There are two possibilities for C: it may contain a greatest element (if a finite stream then C is finite, if an infinite stream then C may be either finite or infinite); or C may contain no greatest element, (in which case it is an infinite chain of finite streams). If C contains a greatest element S, then $Op^{\omega}(S_I) \leq Op^{\omega}(S)$ for all $S_I \in C$ (by isotonicity of Op^{ω} if S is finite, by definition of $Op^{\omega}(S)$ for infinite S). Thus $Op^{\omega}(S)$ is the greatest element of $Op^{\omega}(C)$ and hence its supremum.

If C contains no greatest element, C will be an infinite subchain of *Max-chain*(S) for some infinite S. Since *Cpo-streams* is discrete and since C is infinite, no finite element is an upper bound for C, thus $S = \bigsqcup C$. By definition of $Op^{\omega}(S)$ for infinite S, $Op^{\omega}(S_I) \leq Op^{\omega}(S)$ for all $S_I \in C$. Now the set $Op^{\omega}(C)$ must be a chain because C is a chain and Op^{ω} is isotone. If $Op^{\omega}(C)$ is an infinite chain then it has no finite upper bound so it has the (unique) infinite upper bound $Op^{\omega}(S)$. But there are no other upper bounds so $Op^{\omega}(S) = \bigsqcup Op^{\omega}(C)$. If $Op^{\omega}(C)$ is a finite chain then it has a greatest element $S_G = \bigsqcup Op^{\omega}(C)$. By isotonicity of Op^{ω}, there must exist $S_M \in C$ such that $Op^{\omega}(S_I) = S_G$ for all $S_I \geq S_M$ (obvious for $S_I \in C$, also true for $S_I \in$ *Max-chain*(S)). Thus, $S_G = \bigsqcup Op^{\omega}(\textit{Max-chain}(S)) = Op^{\omega}(S) = \bigsqcup Op^{\omega}(C)$. ⊠

Therefore, we have proved that for any chain C, $Op^{\omega}(\bigsqcup C) = \bigsqcup Op^{\omega}(C)$, so that Op^{ω} is continuous. This applies in the obvious manner to multi-argument operators. Note that in the case of multi-argument operators the order of taking suprema does not matter because the operations of completing a poset and extending an isotone function [Mar-76, Mar-77] give results unique up to isomorphism.

Solvability of First Order Fixed-point Equations

We have now shown that the DFPL primitive operators are isotone on streams, and that we can extend any isotone function on streams to a continuous function on streams by defining its behavior on infinite streams as above. Thus DFPL primitive operators may all be extended to be continuous functions from streams to streams, or more generally, from Cartesian products of streams to (Cartesian products of) streams. Now it is known that any system of equations involving only continuous functions over complete posets have a minimal fixed point solution [Mar-76, Mar-77].

Now any DFPL program graph that includes only primitive operators and no recursion can be converted to an equivalent system of equations. Recall that a program graph corresponds to a set of equations in which each data path corresponds to an equation variable and each operator instance to a function instance. By use of the copy rule, a program graph containing usages of defined operators can be expanded into a graph containing only primitives and thence into a (large) system of equations. Therefore, any such DFPL program has a minimal fixed point solution, that is, an assignment of streams to the data paths which are the overall result of "running" that program (perhaps forever) starting with empty data paths.

Note that the solution obtained is a configuration of data streams and thus represents a particular result of applying the function represented by the program, rather than the function itself. For this reason, we call this a first order fixed-point.

Examples of First Order Fixed-points

Figure 4.1 shows a simple DFPL program with a loop, for which we will compute a first order fixpoint. The *Hold* operator is as discussed earlier. The Every-other operator delivers as output every other element of its input stream. For example $\mathit{Every\text{-}other}(\langle A , B , C , D , E , \ldots\rangle) = \langle A , C , E , \ldots\rangle$. We will not explore the innards of this operator, they are not germane to the fix-point computation. To solve this loop, we cut it at the point labeled X, then we solve the equation $X = \mathit{Every\text{-}other}(\mathit{Hold}_B(\mathit{Hold}_A(X)))$. We do this by applying the standard fix-point rule, computing $\bigsqcup\{\bot, F(\bot), F(F(\bot)), \ldots\}$.

Proceeding by this rule we start with $\mathit{Hold}_A(\langle\,\rangle) = \langle A\rangle$, $\mathit{Hold}_B(\langle A\rangle) = \langle B , A\rangle$ and $X_1 = \mathit{Every\text{-}other}(\langle B , A\rangle) = \langle B\rangle$. Note that this is the first approximation to X, not the first element of X which would be denoted X^1. The second approximation is $X_2 = \mathit{Every\text{-}other}(\mathit{Hold}_B(\mathit{Hold}_A(X_1))) = \mathit{Every\text{-}other}(\mathit{Hold}_B(\mathit{Hold}_A(\langle B\rangle))) = \mathit{Every\text{-}other}(\langle B , A , B\rangle) = \langle B , B\rangle$. The third approximation is $X_3 = \mathit{Every\text{-}other}(\mathit{Hold}_B(\mathit{Hold}_A(X_2))) = \mathit{Every\text{-}other}(\mathit{Hold}_B(\mathit{Hold}_A(\langle B , B\rangle))) = \mathit{Every\text{-}other}(\langle B , A , B , B\rangle) = \langle B , B\rangle$. Thus we have converged after three iterations ($X = X_3 = X_2$). We can also derive the fix-point values of Y and Z. To wit, $Y = \mathit{Hold}_A(X) = \langle A , B , B\rangle$ and $Z = \mathit{Hold}_B(Y) = \langle B , A , B , B\rangle$.

We could equally well have cut the loop at Y or Z. Then we would have solved $Y = \mathit{Hold}_A(\mathit{Every\text{-}other}(\mathit{Hold}_B(Y)))$ or $Z = \mathit{Hold}_B(\mathit{Hold}_A(\mathit{Every\text{-}other}(Z)))$ respectively. Either of these approaches would have given the same results for X, Y and Z.

It is very important to remember that the iterations involved in computing a (first order) fix-point are not the same as the iterations implied by *executing* a DFPL program loop. In computing the fix-point, we are "standing outside of time" and considering the data streams as wholes, whereas in executing the program loop, we are observing the data streams develop within time. This is analogous to the solution of equations in physics: the iterations necessary to solve a dynamical equation do not take place within the time expressed by that equation.

-5-
A Partial Order for Non-Determinacy

Introduction

We have seen that streams of data, partially ordered by the prefix relation, form a domain upon which determinate DFPL operators are continuous functions, so that the function computed by a DFPL program may be determined by means of function composition and computation of fixed-points. Our task now is to find a domain suitable to both determinate and non-determinate operators, that is, a domain in which both kinds of operators may be cast as continuous functions. Part of our task however, is to formulate the domain and functions in such a way as to be compatible with the determinate formulation. That is, there must be a morphism from the general system to the determinate one, mapping the determinate functions in the general system to corresponding functions in the determinate system, and mapping "determinate" elements of the general domain onto corresponding streams in the determinate domain (*Cpo-streams*).

Just as determinate DFPL programs may be viewed as functions on input streams and output streams, it is reasonable to view non-determinate programs as relations from input streams to output streams. Unfortunately, if we take this point of view, we lose the fixed-point theory which is based on continuous functions (although we still have a useful notion of composition for relations). The way out of this problem is to apply the well known "functor" which transforms relations on sets of objects into "equivalent" functions on sets of sets of those objects. Therefore, for the rest of this dissertation we shall characterize non-determinate programs as functions from sets of input streams to sets of output streams. Each stream in the set corresponds to one possible execution Each possible execution of a non-determinate program causes a particular stream, chosen from the set of streams, to appear on a particular data path. If the program is determinate, then only one stream can appear, so the set is a singleton. Thus the natural map between the determinate and non-determinate semantics involves mapping a stream to the singleton set containing that stream.

Counterexamples to "Posets" on Simple Sets of Streams

The exact choice of what kind of set of streams, as we shall see, is crucial to the formulation of a reasonable denotational semantics of DFPL. The obvious choice of a set of streams is just that, a set of streams. If we use this as our domain, the question is what partial order is suitable. The obvious choice for a partial order on sets is the inclusion relation, which is even chain complete. A moments thought, however, shows us that this is unsuitable in that it does not reduce, when applied to singleton sets, to the prefix order on streams. For example, the stream $\langle A \rangle$ is a prefix of the stream $\langle A , B \rangle$, but the singleton set $\{\langle A \rangle\}$ is not a subset of the singleton set $\{\langle A , B \rangle\}$. Thus the subset relation is not a compatible partial order.

To have a compatible partial order, the relation between two singleton sets of streams must reduce to the prefix relation on those two streams. The obvious extension of this to non-singleton sets is to say that streams in the first set are matched with streams in the second and the first set is less than the second iff the prefix relation holds on all the matched streams. Furthermore, our intuition tells us that one set of streams can be "≤" than another in two ways: first, as indicated above, the streams in one may be prefixes of the streams in the other; second, the bigger set may simply contain more streams.

This suggests the following attempt at a partial order, a set of streams Ss_1 is "≤" than a set Ss_2 iff for all streams S_1 in Ss_1, there exists a stream S_2 in Ss_2 such that S_1 is a prefix of S_2. Unfortunately, this is not a partial order but only a quasi-order, since it does not obey the antisymmetry rule. Consider $Ss_1 = \{\langle A \rangle , \langle A , A \rangle\}$ and $Ss_2 = \{\langle A , A \rangle\}$ Here we have both $Ss_1 \leq Ss_2$ and $Ss_2 \leq Ss_1$ but clearly $Ss_1 \neq Ss_2$. One way around this difficulty is to form the equivalence classes of sets of streams which are both "≤" and "≥" to one another. This constructs a "quotient" system in which "≤" is guaranteed to be a true partial order. However, in this quotient domain the semantic equations can only be solved to yield equivalence classes, (i.e. sets of "equivalent" sets of streams) which might not be enough detail for our needs.

We now observe that the trouble with the previous alleged partial order was that it allowed us to match two different streams in Ss_1 with a single stream in Ss_2. Also, our intuition tells us that each element in a set of streams corresponds to a different execution of the program, and since programs should be isotone functions, feeding a program a "bigger" input should not reduce the possible executions. That is, it should not be the case that $\{\langle A \rangle, \langle A , A \rangle\} \leq \{\langle A \rangle\}$. In response to these points, we make another attempt at a partial order: $Ss_1 \leq Ss_2$ iff there is an injective map from Ss_1 to Ss_2 such that each stream in Ss_1 is a prefix of its image in Ss_2. Unfortunately, this too turns out to be only a quasi-order: consider the infinite sets $Ss_1 = \{\langle A \rangle, \langle A , A , A \rangle, \langle A , A , A , A , A \rangle, \ldots\}$ and $Ss_2 = \{\langle A , A \rangle, \langle A , A , A , A \rangle, \langle A , A , A , A , A , A \rangle, \ldots\}$. We can match $\langle A \rangle$ with $\langle A , A \rangle$, $\langle A , A , A \rangle$ with $\langle A , A , A , A \rangle$ etc., discovering that $Ss_1 \leq Ss_2$, or we can match $\langle A , A , A \rangle$ with $\langle A , A \rangle$, $\langle A , A , A , A , A \rangle$ with $\langle A , A , A , A \rangle$ etc. (omitting $\langle A \rangle$) and find that $Ss_1 \geq Ss_2$. But this would imply that Ss_1 is equivalent to Ss_2, which is unreasonable, since they have *no* elements in common. It does not help to demand that the map from one set of streams to another be bijective, since then sets of unequal (finite) size would be incomparable.

The following scenario suggests that we wish Ss_1 to be strictly less than Ss_2. Consider a non-determinate program that operates as follows: it produces an indeterminate, but even (including zero), number of A's on its output port, then copies its input symbols to that output port. Therefore, when presented with the input stream B, its output is $\langle B \rangle$ or $\langle A , A , B \rangle$ or $\langle A , A , A , A , B \rangle$ etc.; when presented with the input stream $\langle A \rangle$, its output is $\langle A \rangle$ or $\langle A , A , A \rangle$ or $\langle A , A , A , A , A \rangle$ etc.; and when presented with $\langle A , A \rangle$, its output is $\langle A , A \rangle$ or $\langle A , A , A , A \rangle$ or $\langle A , A , A , A , A , A \rangle$ etc. More precisely, when applied to the input set $\{\langle A \rangle\}$, it produces Ss_1 above, and when applied to the input set $\{\langle A , A \rangle\}$, it produces Ss_2. Since our intuition tells us that the set $\{\langle A \rangle\}$ is strictly less than $\{\langle A , A \rangle\}$, and since we wish all DFPL programs to be isotone, we must say that the output set Ss_1 is strictly less than Ss_2, for they are clearly not equal.

There are a number of other possible contenders for a partial order on sets of streams. One which actually is a partial order, and not merely a quasi-order, defines $Ss_1 \leq Ss_2$ iff there exists an injective map from Ss_1 to Ss_2 such that each element of Ss_1 is less than its image in Ss_2, such that the image of the map is a closed below subset of Ss_2 (i.e. whenever X is in the subset, so are all $Y \leq X$), and such that the map is co-isotone (i.e. $F(X) \leq F(Y)$ implies $X \leq Y$). Unfortunately, not all DFPL operators are isotone in this partial order, so it too is unsuitable for our purposes. In fact, we conjecture that there is no suitable partial order (if we restrict ourselves to plain sets of streams) which does not require using equivalence classes of sets of streams, which reduces to the prefix order on singleton sets, and in which all primitive operators are isotone functions.

Another Problem with Simple Sets of Streams

Consider the DFPL program in Figure 5.1. It consists of two two-input *Arbiter*'s whose outputs are connected to the inputs of a simple primitive *Add* operator. If the input streams to the *Arbiter*'s are, as illustrated, the singletons ⟨2⟩, ⟨3⟩, ⟨4⟩ and ⟨5⟩, then the *Arbiter*'s outputs are the sets {⟨2 , 3⟩, ⟨3 , 2⟩} and {⟨4 , 5⟩, ⟨5 , 4⟩}, which are also the inputs to the sides of the adder as shown. Now, since any determinate operator must, if confronted with sets of input streams, combine each stream element of each set with every element of every other (i.e. it must operate on the Cartesian product of its input sets) the output of the adder must be the set {⟨6 , 8⟩, ⟨7 , 7⟩, ⟨8 , 6⟩}. Note that the stream ⟨7 , 7⟩ would be generated twice but would only appear once, because the output is a *set*.

Now consider the DFPL program in Figure 5.2. It consists of a single two input *Arbiter* whose output is connected to *both* inputs of the simple primitive *Add* operator. If the input streams to the *Arbiter* are, as illustrated, the singletons ⟨2⟩ and ⟨3⟩, then the *Arbiter*'s output is the set {⟨2 , 3⟩, ⟨3 , 2⟩}, which is also the input to both sides of the adder as shown. Now, by the Cartesian product rule we used above, the output of the adder would be {⟨4 , 6⟩, ⟨5 , 5⟩, ⟨6 , 4⟩}. This, unfortunately, is a result which the operational semantics of DFPL contradicts — the stream ⟨5 , 5⟩ can never be a result of this program. The *Arbiter* either outputs ⟨2 , 3⟩ or ⟨3 , 2⟩, it can never output both together nor can it output their average!

This example demonstrates that simple sets of streams is not an adequate basis for the denotational semantics of even very simple programs not involving fix-points. Such simple sets just do not contain enough information to allow such programs as Figure 5.1 to be distinguished from programs like Figure 5.2. In particular, the adder operator has no way of knowing whether its inputs came from the same or different *Arbiter*'s. Thus we feel justified in searching for a somewhat more complicated basis. We must incorporate in each stream an indication of how it may have been arbitrated.

Sets of Tagged Streams of Data

It is possible to obtain a straightforward partial order by considering sets of tagged streams of data. Each datum in each stream in the set has associated with it zero or more tags, each of which identifies the sequence of arbitrary choices made by a non-determinate operator which contributed to the existence of that datum in that stream. Sets of tagged streams are constrained in the following two ways. A later datum may never be the result of fewer non-determinate choices than an earlier datum, and no stream is merely an approximation to another.

Two sets are compared by matching each stream in the first set with a stream in the second set such that the first stream is a prefix of the second stream. The prefix relation used here is the same as that used in *Cpo-streams*, except that the items in tagged-stream are pairs of *Data* and *Tag-set*, rather than merely *Data*. However, all the relevant properties apply to tagged-streams. This relation may be shown to be a true partial ordering of sets of tagged streams, and the resulting poset is chain complete if infinite streams and sets are admitted.

Each instance of an *Arbiter* in a DFPL program is uniquely identified by its *Arbiter-name*, an element of set with equality. Remember that each recursion level generates new instances of its operators. A *Choice-sequence* of an *Arbiter* is an empty, finite or infinite sequence of integers, chosen from range 0 through *Number-of-input-ports* $-$ 1. A *Choice-sequence* represents, in order, the non-determinate choices made by an *Arbiter*. A *Tag* is a pair ⟨*Arbiter-name*, *Choice-sequence*⟩, and represents the choices made by a particular *Arbiter*. A *Tag-set* is an empty or finite set of *Tags* such that no two elements have the same *Arbiter-name* component. A tag-set represents the non-determinate choices made by a set of *Arbiters*. The restriction that no two elements have the same *Arbiter-name* insures that no tag-set represents that an *Arbiter* has made self-contradictory choices. A tag-set Ts_2 is said to be an *Extension* of a tag-set Ts_1 iff there is an injective map from Ts_1 to Ts_2 such that for each element of Ts_1, the *Arbiter-name* of that element is the same as the *Arbiter-name* of its image, and the *Choice-sequence* of the element is a prefix of the *Choice-sequence* of its image. More precisely, we say that:

```
$Tgs \sqsubseteq Tgsx \equiv$
  $\exists Map : Tgs \rightarrow Tgsx:$
    $Injective(Map) \wedge$
      $\forall Tg \in Tgs:$
        $Arbiter\text{-}name(Tg) = Arbiter\text{-}name(Map(Tg)) \wedge$
        $Choice\text{-}sequence(Tg) \leq Choice\text{-}sequence(Map(Tg))$
```

A *Datum* is an element of some set of data. All data are assumed to be incomparable from the denotational point of view. A *Tagged-stream* is an empty, finite or infinite sequence of pairs of the form (*Datum*, *Tag-set*) which obeys the tag-set extension rule. This rule demands that the tag-set component of any element in the tagged-stream is an extension of the tag-set component of all elements preceding it in that tagged-stream. This insures that no datum is the result of fewer non-determinate choices than a datum which occurred earlier in that tagged-stream. A *Tagged-stream-set* is a non-empty set of tagged-streams which is *Prefix-reduced*. This means that no tagged-stream is a strict prefix of any other tagged-stream in the tagged-stream-set. This insures that no tagged-stream is merely an approximation to another in the same tagged-stream-set.

For example, the result of supplying a two port (non-determinate) *Arbiter* with the (determinate) inputs $\{\langle A\rangle\}$ and $\{\langle B\rangle\}$ yields as output the (non-determinate) tagged-stream-set $\{\langle A_0, B_{01}\rangle, \langle B_1, A_{10}\rangle\}$. The result of passing that set through a (determinate) operator which throws away input data until an *A* appears, whereupon it copies the rest of the stream to its output port, is $\{\langle A_0, B_{01}\rangle, \langle A_{10}\rangle\}$.

The partial order on tagged-stream-sets may now be defined. A tagged-stream-set Tss_1 is $\leq$ a tagged-stream-set Tss_2 iff there exists an injective map from Tss_1 to Tss_2 such that each element of Tss_1 is a prefix of its image in Tss_2. Note that this implies that the cardinality of Tss_1 is no bigger than that of Tss_2. Also note that this is equivalent to saying that $Tss_1 \leq Tss_2$ iff for all elements of Tss_1, there exists an element of Tss_2 of which it is a prefix. This may be shown as follows. If *Sa*, *Sb* and *Sc* are streams such that *Sa* is a prefix of *Sc* and *Sb* is a prefix of *Sc*, then either *Sa* is a prefix of *Sb* or *Sb* is a prefix of *Sa*. This implies that if there exists an element S_2 of Tss_2 of which an element

Sa_1 of Tss_1 is a prefix, then there is no other element Sb_1 of Tss_1 which is also a prefix of S_2, otherwise, either Sb_1 would be a prefix of Sa_1 or vice versa, and we disallow this in the definition of tagged-stream-set.

The map which takes a stream S into a tagged-stream-set $Tss = \{St\}$ such that $St^I = \langle S^I, \{\ \}\rangle$ for all $I \in Dom(S)$, is an monomorphism of posets. This will become clear in the next two sections, justifying our implied claim of a non-determinate domain which is compatible with the domain *Cpo-streams*. We shall see later that the *Tags* have another use besides allowing the definition of a compatible partial order.

The way that the tags force the tagged-streams of one tagged-stream-set to be compared with particular tagged-streams of the other tagged-stream-set is reminiscent of the "arrows" in Lehman's categories which he uses to model domains for non-deterministic fixed-point semantics [Leh-76].

Proof of Partial Order

Theorem 5.1: The relation "$\preccurlyeq$" is a partial order.

To prove that "$\preccurlyeq$" is a partial order on tagged-stream-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-stream is a prefix of itself, we have $Tss_1 \preccurlyeq Tss_1$.

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 "$\preccurlyeq$" 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-stream-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 "$\trianglelefteq$" is antisymmetric. ▣

Proof of Completeness

Theorem 5.2: The partial order "$\trianglelefteq$"is (countable) chain complete.

To prove this, we must show that any countable chain has a supremum. Let $\{Tss_1 \trianglelefteq Tss_2 \trianglelefteq Tss_3 \trianglelefteq \ldots\}$ be such a countable chain, and let $\{M_1, M_2, \ldots\}$ be the associated sequence of injective maps which specify the relations (i.e. $M_1 : Tss_1 \rightarrow Tss_2$, $M_2 : Tss_2 \rightarrow Tss_3$ etc.). Let S be an element of Tss_N, then the set $\{S, M_N(S), M_N \circ M_{N+1}(S), \ldots\}$ forms a chain under the prefix order. Since *Cpo-streams* is chain complete, this set has a supremum which we call *S-sup*. Call the set of all such suprema *Tss-sup*. Since all the *M*'s are injective, and each *Tss* is prefix-reduced, we apply Lemma 4.1 to deduce that 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 *S-sup*, the supremum of its chain. The suprema of all distinct chains are themselves distinct (by Lemma 5.1 and the assumption that *Tss*'s are prefix-reduced) because each chain has at least one non-supremal element not the other chain. Then we have that $Msup_N : Tss_N \rightarrow Tss\text{-}sup$ is an injective map which establishes that $Tss_N \trianglelefteq Tss\text{-}sup$. 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 $S\text{-}ub \lessdot S\text{-}sup$, contradicting the fact that *S-sup* was the supremum of that chain. In the second case, there would be a chain of elements from the *Tss*'s which had no supremum in *Tss-ub*, hence *Tss-ub* could not even be an upper bound. Therefore, we may conclude that *Tss-sup* is indeed the supremum of the *Tss*'s.

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

If one tagged-stream, Ts_1, were a strict prefix of another, Ts_2, then all the elements of the chain of which Ts_1 was the supremum would be in the chain of Ts_2, hence Ts_1 could not be their supremum.

If the tag-set extension property were not obeyed, then there would exist a tagged-stream *Ts-sup* in *Tss-sup* such that $Tag\text{-}set(Ts\text{-}sup^K)$ did not extend $Tag\text{-}set(Ts\text{-}sup^J)$ (where $J \leq K$). But, since *Tss-sup* is the supremum of its chain of *Tss*'s, there would exist some Tss_N which contained a tagged-stream $Ts \leq Ts\text{-}sup$ such that $Ts^J = Ts\text{-}sup^J$ and $Ts^J = Ts\text{-}sup^J$ contradicting the tag-set extension property assumed for the *Tss*'s.

Therefore the *Tss-sup* is a proper tagged-stream-set and is the supremum of the *Tss*'s, which means that the set of tagged-stream-sets is a complete poset. ⊠

Satisfaction of Previous Counterexamples

As we have just proved, the set of tagged-stream-sets form a chain complete partially ordered set (not merely a quasi-ordered set). Also, we have shown how the set of tagged-stream-sets is compatible with streams under the map which takes a stream S into the singleton set consisting of the tagged-stream whose *Data* are the same as S, and whose tag-sets are empty. Therefore we have satisfied the two generic counterexamples.

The specific counterexample involved a non-determinate program which had two states: produce A's (state 0) and copy input (state 1). Using the history of these states as the *Choice-sequence* attached to each output datum (and since there only need be one *Arbiter*, omitting the *Arbiter-name* and set brackets from the tag-set), we get the following specification of the program: an input of $\{\langle A \rangle\}$ gives rise to the output:

$$\{\langle A_1 \rangle, \langle A_0, A_{00}, A_{001} \rangle, \langle A_0, A_{00}, A_{000}, A_{0000}, A_{00001} \rangle\}$$

whereas an input of $\{\langle A, A \rangle\}$ gives rise to the output:

$$\{\langle A_1, A_{11} \rangle, \langle A_0, A_{00}, A_{001}, A_{0011} \rangle, \langle A_0, A_{00}, A_{000}, A_{0000}, A_{00001}, A_{000011} \rangle\}$$

The rules for match tagged-streams in one tagged-stream-set with those in another make it quite clear that the first output is $\preceq$ the second.

Referring back to Figure 5.2 now, we see that the *Arbiter*'s output streams would be tagged as follows: $\{\langle 2_0 , 3_{01}\rangle , \langle 3_1 , 2_{10}\rangle\}$. Let us adopt a modified Cartesian product rule, to be detailed in the next chapter, that tuples of input streams are combined by an operator only if their *Tags* are *Consistent*. Then the output of the adder would be the tagged-stream-set $\{\langle 4_0 , 6_{01}\rangle , \langle 6_1 , 4_{10}\rangle\}$. The stream $\langle 5 , 5\rangle$ cannot appear at all in the output set because its first element would have to be tagged both 0 and 1. This is impossible since it would mean that the *Arbiter* made two mutually exclusive decisions at once.

In summation, we have constructed a domain for non-determinate semantics that satisfies all the objections we discovered to the earlier approaches.

-6-
Semantics of Non-Determinate DFPL Programs

Overview

In this chapter we develop the fixed-point semantics of full DFPL with both the determinate and non-determinate primitives. We have shown, in Chapter 5, that the domain of *Tagged-stream-sets* is suitable for fixed-point solutions of programs. We must now show that the DFPL operators are continuous on this domain. To do this, we first develop some helper functions on tag-sets, then we show how the determinate operators are extended to tagged-stream-sets, then we can show that the determinate operators are continuous. Next we precisely define the non-determinate *Arbiter* and prove that it is continuous also. We can therefore deduce that all (recursion free) DFPL programs have a well defined behavior no matter what inputs they receive. We conclude with an example of a simple fixed-point computation of a non-determinate program containing a loop.

Notation

The notation used in this chapter is somewhat complicated and thus is outlined here. Variables are written out programming style (i.e. multi-letter abbreviations) as in earlier chapters, but there are more possibilities. Variables consist of a head (usually an abbreviation), which connotes their domain, an optional body followed by an optional tail, which identify the particular variable, and an optional subscript, which identifies one of a group of similar variables. Commonly appearing heads are: *Ts* for a tagged-stream, *Tss* for a tagged-stream-set, *Tg* for a tag and *Tgs* for a tag-set. The common optional bodies are: *-a-*, *-b-*, *-c-* and *-d-*, where *-a-* and *-b-* connote arbitrary distinct variables, *-c-* and *-d-* usually connote control and data inputs respectively, and lack of a body usually connotes an output variable. An tail *-x* usually connotes extension of a stream or set, that is $Ts \leq Tsx$, $Tgs \sqsubseteq Tgsx$ etc. Some examples are: *Tss* for an output tagged-stream-set, $Tsdx_I$ for the *I*th extended data input tagged-stream, *Tgsx* for an extended tag-set, and *Tga* for an arbitrary tag.

In the interest of brevity, we often apply a function to a set of arguments without writing it out explicitly. For example, if $F: X \times Y \rightarrow Z$, we write $F(Xs, Ys)$ (where $Xs \subseteq X$ and $Ys \subseteq Y$) instead of $\{F(Xa, Ya) \mid (Xa, Ya) \in Xs \times Ys\}$. In general, if a function takes an element of some domain as an argument, we may apply that function to a set of such arguments implying that the appropriate set of results is denoted. Note that the original domain may have sets as elements, in which case we would apply the function to a set of such sets.

Tag-set Functions and their Properties

In order to define the extensions of the determinate operators ($Hold^{\xi}$, Sop^{ξ}, $Oswitch^{\xi}$ and $Iswitch^{\xi}$), we define some helpful auxiliary functions. First of all, we define some access functions which allow us to take components of *Tagged-streams* and *Tags* in a clear manner:

$$Datum(Ts) = Ts^1$$

$$Tag\text{-}set(Ts) = Ts^2$$

$$Arbiter\text{-}name = Tag^1$$

$$Choice\text{-}sequence = Tag^2$$

The next function, *Consistent-tags*, is a predicate which is true of unions of tag-sets which are consistent, that is, tag-sets which do not contain *Tags* with the same *Arbiter-name* but *Choice-sequences* which are not prefixes of each other (i.e. *Choice-sequences* which do not form a chain):

$$\begin{aligned}
&Consistent\text{-}tags(Tgs_1, \ldots, Tgs_N) \equiv \\
&\quad \forall\, Tga, Tgb \in \textstyle\bigcup_{I \leq N} Tgs_I: \\
&\qquad Arbiter\text{-}name(Tga) = Arbiter\text{-}name(Tgb) \rightarrow \\
&\qquad\quad Choice\text{-}sequence(Tga) \leq Choice\text{-}sequence(Tgb) \vee \\
&\qquad\quad Choice\text{-}sequence(Tga) \geq Choice\text{-}sequence(Tgb)
\end{aligned}$$

The last helpful auxiliary function is related to the previous. It is the *Merge-tags* function, which merges the *Tags* in a consistent tag-set union to yield a tag-set which contains, for each *Arbiter-name*, the maximal *Choice-sequence* from the input *Tags*:

$$
\begin{aligned}
&\textit{Merge-tags}(Tgs_1, \ldots, Tgs_N) = \\
&\quad \{\, Tg \in \textstyle\bigcup_{I \le N} Tgs_I \mid \\
&\qquad \forall\, Tga \in \textstyle\bigcup_{I \le N} Tgs_I : \\
&\qquad\quad \textit{Arbiter-name}(Tg) = \textit{Arbiter-name}(Tga) \Rightarrow \\
&\qquad\quad \textit{Choice-sequence}(Tg) \nless \textit{Choice-sequence}(Tga) \,\}
\end{aligned}
$$

The *Merge-tags* function is used to generate the tag-sets for the outputs of operators given their input tag-sets. The *Consistent-tags* function is used to assure that operators do not process any input streams which are inconsistent with each other (cf. Chapter 5, especially Figure 5.2 and related text).

Lemma 6.1: Both *Consistent-tags* and *Merge-tags* are commutative and associative functions. That is, *F-tags*(*Tgsa*, *F-tags*(*Tgsb*, *Tgsc*)) = *F-tags*(*Tgsa*, *Tgsb*, *Tgsc*) = *F-tags*(*Tgsc*, *F-tags*(*Tgsa*, *Tgsb*)) = (where *F-tags* is *Consistent-tags* or *Merge-tags*).

This follows directly from the definitions.

Lemma 6.2: If $Tgs \sqsubseteq Tgsx$ then *Consistent-tags*(*Tgs*, *Tgsx*) is ***True***. This too follows directly from the definitions of "$\sqsubseteq$" and *Consistent-tags*.

Lemma 6.3: If *Consistent-tags*($Tgs_1, \ldots, Tgs_I, \ldots, Tgs_N$) is ***False***, then for any I and any $Tgsx_I$ such that $Tgs_I \sqsubseteq Tgsx_I$, *Consistent-tags*($Tgs_1, \ldots, Tgsx_I, \ldots, Tgs_N$) is also ***False***. Expanding the definition of *Consistent-tags*($Tgs_1, \ldots, Tgs_I, \ldots, Tgs_N$) we find that it is ***False*** iff:

$$
\begin{aligned}
&\exists\, Tga, Tgb \in \textstyle\bigcup_{J \le N} Tgs_J : \\
&\textit{Arbiter-name}(Tga) = \textit{Arbiter-name}(Tgb) \wedge \\
&\textit{Choice-sequence}(Tga) \nleq \textit{Choice-sequence}(Tgb) \wedge \\
&\textit{Choice-sequence}(Tga) \ngeq \textit{Choice-sequence}(Tgb)
\end{aligned}
$$

Now recall the definition of "$\sqsubseteq$":

$$
\begin{aligned}
&Tgs_I \sqsubseteq Tgsx_I \equiv \\
&\quad \exists Map: Tgs_I \rightarrow Tgsx_I: \\
&\qquad Injective(Map) \wedge \\
&\qquad\quad \forall Tg \in Tgs_I: \\
&\qquad\qquad Arbiter\text{-}name(Tg) = Arbiter\text{-}name(Map(Tg)) \wedge \\
&\qquad\qquad Choice\text{-}sequence(Tg) \leq Choice\text{-}sequence(Map(Tg))
\end{aligned}
$$

Now if neither the Tga or the Tgb which falsified $Consistent\text{-}tags(Tgs_1, \ldots, Tgs_I, \ldots, Tgs_N)$ is an element of Tgs_I, then $Consistent\text{-}tags(Tgs_1, \ldots, Tgsx_I, \ldots, Tgs_N)$ is trivially ***False*** also. If, however, either Tga or Tgb is an element of Tgs_I then its image in $Tgsx_I$ under Map serves as a counterexample to $Consistent\text{-}tags(Tgs_1, \ldots, Tgsx_I, \ldots, Tgs_N)$ (due to the properties of "$\leq$"). $\boxtimes$

Lemma 6.3 assures that the tagged-streams which result from the operators defined recursively in the next section are well behaved in the sense that if the recursion is terminated by $Consistent\text{-}tags(\ldots)$ becoming ***False***, there are no elements farther down some input stream which might contribute to the output, if only they could be reached. That is, the recursive definitions do not disallow any realizable behavior.

Lemma 6.4: If $Tgsx_I \sqsubseteq Tgs_I$ then $Merge\text{-}tags(Tgs_1, \ldots, Tgs_I, \ldots, Tgs_N) \sqsubseteq Merge\text{-}tags(Tgs_1, \ldots, Tgsx_I, \ldots, Tgs_N)$. Recalling that:

$$
\begin{aligned}
&Merge\text{-}tags(Tgs_1, \ldots, Tgs_I, \ldots, Tgs_N) = \\
&\quad \{ Tg \in \cup_{J \leq N} Tgs_J \mid \\
&\qquad \forall Tga \in \cup_{J \leq N} Tgs_J: \\
&\qquad\quad Arbiter\text{-}name(Tg) = Arbiter\text{-}name(Tga) \Rightarrow \\
&\qquad\quad Choice\text{-}sequence(Tg) \not< Choice\text{-}sequence(Tga) \}
\end{aligned}
$$

We immediately derive that:

$$
\begin{aligned}
&Merge\text{-}tags(Tgs_1, \ldots, Tgsx_I, \ldots, Tgs_N) = \\
&\quad \{ Tgx \in Tgs_I \cup \cup_{J \neq I} Tgs_J \mid \\
&\qquad \forall Tga \in Tgs_I \cup \cup_{J \neq I} Tgs_J: \\
&\qquad\quad Arbiter\text{-}name(Tgx) = Arbiter\text{-}name(Tga) \Rightarrow \\
&\qquad\quad Choice\text{-}sequence(Tgx) \not< Choice\text{-}sequence(Tga) \}
\end{aligned}
$$

Let $Mtg = Merge\text{-}tags(Tgs_1, \ldots, Tgs_I, \ldots, Tgs_N)$ and $Mtgx = Merge\text{-}tags(Tgs_1, \ldots, Tgsx_I, \ldots, Tgs_N)$. To show that $Mtg \sqsubseteq Mtgx$ we must construct a map $Mtmap: Mtg \rightarrow Mtgx$ which satisfies the definition of "$\sqsubseteq$" above. Let $Tg \in Mtg$, $Tgx \in Mtgx$ and $Tgx = Mtmap(Tg)$. Now define $Mtmap$ as follows, if $Tg \in Tgs_J$ where $J \neq I$, then $Tgx = Tg$, but if $Tg \in Tgs_I$ then $Tgx = Map(Tg)$ where Map is the map which makes $Tgs_I \sqsubseteq Tgsx_I$ as above. Obviously $Mtmap$ satisfies the second (quantified) part of the definition of "$\sqsubseteq$" since both Map and the identity do, so we only need show that $Mtmap$ is injective. Let $Tga \neq Tgb$ be arbitrary elements of Mtg and let $Tgax$ and $Tgbx$ be their images under $Mtmap$. Note that neither $Tga \leq Tgb$ nor $Tgb \leq Tga$ can be true because of the definition of Mtg. If neither Tga nor Tgb are in Tgs_I then $Tgax = Tga$ and $Tgbx = Tgb$ so $Tgax \neq Tgbx$ since $Mtmap$ is the identity except on Tgs_I. Similarly, if both Tga and Tgb are in $Tgsx_I$, then $Tgax \neq Tgbx$ because $Mtmap = Map$ when restricted to $Tgsx_I$ and Map is assumed injective. The interesting cases are when $Tga \in Tgs_I$ and $Tgb \notin Tgs_I$ or vice-versa. In the first case, $Tgbx = Tgb$ and $Tgax = Map(Tga)$. But by the assumption that Map shows how $Tgs_I \sqsubseteq Tgsx_I$, we know that $Choice\text{-}sequence(Tga) \leq Choice\text{-}sequence(Tgax)$. By the properties of "$\leq$" and sequences, and by the fact that $Tga \nleq Tgb$ and $Tgb \nleq Tga$, we see that $Tgb \neq Tgax$ and hence that $Tgbx \neq Tgax$. Therefore $Mtmap$ is injective and $Mtg \sqsubseteq Mtgx$. ⊠

Lemma 6.5: If $Tgs \sqsubseteq Tgsx$ then $Merge\text{-}tags(Tgs, Tgsx) = Tgsx$. Substituting in the definition of $Merge\text{-}tags$, we get:

```
Merge-tags ( Tgs , Tgsx ) =
  { Tg ∈ Tgs ∪ Tgsx |
    ∀ Tga ∈ Tgs ∪ Tgsx:
      Arbiter-name ( Tg ) = Arbiter-name ( Tga ) ⇒
      Choice-sequence ( Tg ) ≮ Choice-sequence ( Tga ) }
```

If $Tg \in Tgs$ then $\exists Tgx \in Tgsx: Tg < Tgx$ (because $Tgs \sqsubseteq Tgsx$) so no elements of Tgs contribute themselves to $Merge\text{-}tags(Tgs, Tgsx)$ unless they also are in $Tgsx$. Therefore $Merge\text{-}tags(Tgs, Tgsx) = Tgsx$. ⊠

Lemma 6.6: If $Consistent\text{-}tags(Tgsa, Tgsb)$ then $Tgsa \sqsubseteq Merge\text{-}tags(Tgsa, Tgsb)$, and symmetrically, $Tgsb \sqsubseteq Merge\text{-}tags(Tgsa, Tgsb)$.

Consider all pairs $\langle Tga, Tgb \rangle \in Tgsa \times Tgsb$. If $Arbiter\text{-}name(Tga) \neq Arbiter\text{-}name(Tgb)$ then both get included in $Merge\text{-}tags(Tgsa, Tgsb)$. If $Arbiter\text{-}name(Tga) = Arbiter\text{-}name(Tgb)$ then the one that is the prefix of the other (and one is since the sets are consistent) gets discarded in the construction of $Merge\text{-}tags(Tgsa, Tgsb)$. Since both $Tgsa$ and $Tgsb$ have only one occurrence of each *Arbiter-name*, we can stop after considering each pair, no further prefixing can obtain. Hence each Tga either itself appears in $Merge\text{-}tags(Tgsa, Tgsb)$ or a Tgb of which it is a prefix appears. Therefore, by the definition of "$\sqsubseteq$", we see that $Tgsa \sqsubseteq Merge\text{-}tags(Tgsa, Tgsb)$ and symmetrically. ⊠

Extension of Determinate Operators

The *Hold* operator is the simplest operator to extend to tagged-stream-sets. It is defined as follows:

$$Hold_C^{\xi}(Tss) = \{ Tsa \in Hold_C^{\omega}(Tss) \mid \forall Tsb \in Hold_C^{\omega}(Tss) : Tsa \not< Tsb \}$$

$$\textit{Where } Hold_C^{\omega}(Ts) =$$
$$\textit{If } Dom(Ts) < \omega \textit{ Then } Hold_C(Ts) \textit{ Otherwise } \bigsqcup_{Tsf < Ts} Hold_C(Tsf)$$

$$\textit{Where } Hold_C(Ts) = \langle C, \{\,\} \rangle \oplus Ts$$

That is, $Hold_C^{\xi}(Tss)$ is the set of all tagged-streams from Tss with the additional item "empty-tagged C" attached to the front, and the resulting set is reduced to eliminate strict prefixes. But since Tss is already *Prefix-reduced*, and since the extension of "$\oplus$" is trivial, we can simplify the definition to:

$$Hold_C^{\xi}(Tss) = \{ \langle C, \{\,\} \rangle \oplus Ts \mid Ts \in Tss \}$$

Note that since any tag-set extends the empty tag-set, the tagged-streams in $Hold_C^{\xi}(Tss)$ obey the tag-set extension rule. Note also that the unextended *Hold* function here is very similar to the *Hold* function in Chapter 4.

To simplify the definitions of the remaining DFPL operators, we define once and for all the completion Op^{ω} of an operator Op. Namely:

$$Op^{\omega}(\ldots, Ts_1, \ldots, Ts_N) = \bigsqcup Op(Cs(Ts_1), \ldots, Cs(Ts_N))$$

$$\textit{Where } Cs(Ts) = \\ \quad \textit{If } Dom(Ts) < \omega \textit{ Then } \{Ts\} \\ \quad \textit{Otherwise } \{Tsa \mid Tsa \lessdot Ts\}$$

This definition is the obvious generalization of the one used above in the definition of $Hold_C^\omega$ and is equivalent to the definition used in Chapter 4.

Lemma 6.7: The completion Op^ω of an isotone operator Op, as defined above, is continuous on its *Tagged-stream* arguments.

The single "$\sqcup$" is well defined because the codomain of Op is the set *Tagged-streams* which is ω-chain complete and the set $Op(Cs(Ts_1), \ldots, Cs(Ts_N))$ is directed and of cardinality no bigger than ω [Mar-76, Mar-77]. The proof is thus the obvious generalization of the proof of Theorem 4.3, substituting directed sets for chains and the domain *Tagged-streams* for *Cpo-streams*.

Now we can define the extended Simple Operators. The extended Simple Operator Sop_F^ξ of N arguments and one parameter F (the function to be applied), essentially takes the Cartesian product of its input tagged-stream-sets and applies the stream operator Sop_F to each element thereof to get an output tagged-stream-set, from which strict prefixes are eliminated (*Prefix-reduction*). However, whenever Sop_F finds tag-sets which are mutually inconsistent, it ceases processing that particular N-tuple of input streams, truncating the output stream accordingly. This insures that F is not applied to any data which could not coexist under a particular sequence of non-determinate choices. The precise definitions of Sop_F^ξ and Sop_F are:

$$Sop_F^\xi(Tss_1, \ldots, Tss_N) = \\ \quad \{Tsa \in Sop_F^\omega(Tss_1, \ldots, Tss_N) \mid \\ \qquad \forall\, Tsb \in Sop_F^\omega(Tss_1, \ldots, Tss_N) : Tsa \not\lessdot Tsb\}$$

```
Where $Sop_F(Ts_1, \ldots, Ts_N) =$
  If $\exists I \leq N: Ts_I = \langle\,\rangle$ Then $\langle\,\rangle$
  If Consistent-tags(Tag-set($Ts_1{}^1$), ..., Tag-set($Ts_N{}^1$)) Then
    $\langle F(Datum(Ts_I{}^1), \ldots, Datum(Ts_N{}^1)),$
      Merge-tags(Tag-set($Ts_1{}^1$), ..., Tag-set($Ts_N{}^1$))$\rangle \oplus$
    $Sop_F(\tau\, Ts_1, \ldots, \tau\, Ts_N)$
  Otherwise $\langle\,\rangle$
```

Next we define the extended Outbound Switch operator. The $Oswitch_P{}^{\xi}$ operator takes two arguments: the control tagged-stream-set, *Tssc*, the data tagged-stream-set, *Tssd*, and one parameter, the port number *P*. This parameter is necessary since the Outbound Switch is an *N* output operator, and mathematical notation does not directly allow such functions. $Oswitch_P{}^{\xi}$ essentially applies $Oswitch_P$ to each pair of input tagged-streams in the input tagged-stream-sets, and then *Prefix-reduces* the resulting set. Again, $Oswitch_P$ stops processing its input streams whenever it finds two tag-sets which are inconsistent. The precise definitions of $Oswitch_P{}^{\xi}$ and $Oswitch_P$ are:

```
$Oswitch_P{}^{\xi}(Tssc, Tssd) =$
  $\{ Tsa \in Oswitch_P{}^{\omega}(Tssc, Tssd) \mid \forall Tsb \in Oswitch_P{}^{\omega}(Tssc, Tssd): Tsa \not\prec Tsb \}$

Where $Oswitch_P(Tsc, Tsd) =$
  If $Tsc = \langle\,\rangle \vee Tsd = \langle\,\rangle$ Then $\langle\,\rangle$
  If $Datum(Tsc^1) = P \wedge$ Consistent-tags(Tag-set($Tsc^1$), Tag-set($Tsd^1$)) Then
    $\langle Datum(Tsd^1),$ Merge-tags(Tag-set($Tsc^1$), Tag-set($Tsd^1$))$\rangle \oplus$
    $Oswitch_P(\tau\, Tsc, \tau\, Tsd)$
  If $Datum(Tsc^1) \neq P \wedge$ Consistent-tags(Tag-set($Tsc^1$), Tag-set($Tsd^1$)) Then
    $Oswitch_P(\tau\, Tsc, \tau\, Tsd)$
  Otherwise $\langle\,\rangle$
```

Last we define the extended Inbound Switch operator. The $Iswitch^{\xi}$ operator takes $N+2$ arguments: the control tagged-stream-set, *Tssc*, and $N+1$ data tagged-stream-sets, Tss_I. $Iswitch^{\xi}$ applies *Iswitch* to each $N+2$-tuple from the Cartesian product of its inputs sets. *Iswitch* stops when it finds tag-sets which are not consistent, as usual, but note that *Iswitch* recurs differently than the previous operators; although it always takes "τ" of *Tsc* (the control tagged-stream), it only takes "τ" of the selected data tagged-stream, Ts_I. The precise definitions of $Iswitch^{\xi}$ and $Iswitch$ are:

$$\begin{aligned}
&Iswitch^{\xi}(Tssc, Tss_0, \ldots, Tss_N) = \\
&\quad \{ Tsa \in Iswitch^{\omega}(No\text{-}tags, Tssc, Tss_0, \ldots, Tss_N) \mid \\
&\qquad \forall Tsb \in Iswitch^{\omega}(No\text{-}tags, Tssc, Tss_0, \ldots, Tss_N) : Tsa \nless Tsb \}
\end{aligned}$$

Where $No\text{-}tags = \{\ \}$

$$\begin{aligned}
&\textbf{And } Iswitch(Tgs, Tsc, Ts_0, \ldots, Ts_N) = \\
&\quad \textbf{If } Tsc = \langle\ \rangle \textbf{ Then } \langle\ \rangle \\
&\quad \textbf{If } Datum(Tsc_1) = 0 \wedge Ts_0 = \langle\ \rangle \textbf{ Then } \langle\ \rangle \\
&\qquad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \\
&\quad \textbf{If } Datum(Tsc_1) = N \wedge Ts_N = \langle\ \rangle \textbf{ Then } \langle\ \rangle \\
&\quad \textbf{If } Datum(Tsc_1) = 0 \wedge Consistent\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_0{}^1)) \textbf{ Then} \\
&\qquad \langle Datum(Ts_0{}^1), Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_0{}^1))\rangle \oplus \\
&\qquad Iswitch(Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_0{}^1)), \\
&\qquad\qquad \tau\, Tsc, \tau\, Ts_0, \ldots, Ts_N) \\
&\qquad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \\
&\quad \textbf{If } Datum(Tsc_1) = N \wedge Consistent\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_N{}^1)) \textbf{ Then} \\
&\qquad \langle Datum(Ts_N{}^1), Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_N{}^1))\rangle \oplus \\
&\qquad Iswitch(Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_N{}^1)), \\
&\qquad\qquad \tau\, Tsc, Ts_0, \ldots, \tau\, Ts_N) \\
&\quad \textbf{Otherwise } \langle\ \rangle
\end{aligned}$$

Continuity of the Determinate Operators

Theorem 6.1: The extensions of determinate operators, as defined above, are continuous functions in all of their tagged-stream-set arguments.

Let F^{ω} be a function on tagged-streams which is continuous and thus isotone, and let $\{Tss_1, Tss_2, \ldots, Tss\}$ be a chain whose supremum is Tss. (Although F may be a function of several arguments, we are considering continuity in one argument at a time so for brevity we elide the others and write only $F(X)$.) Consider the sequence of image sets: $\{F^{\omega}(Tss_1), F^{\omega}(Tss_2), \ldots, F^{\omega}(Tss)\}$; note that this is not the extension of F^{ω} as above, just the normal application of a function to a set of arguments. If $\{Ts_K, Ts_{K+1}, \ldots, Ts\}$, where $Ts_I \in Tss_I$, is a chain whose supremum is Ts, then $\{F^\omega(Ts_K), F^\omega(Ts_{K+1}), \ldots, F^\omega(Ts)\}$ is a chain whose supremum is $F^\omega(Ts)$. (We start the chain with K instead of 1 because the cardinality $|Tss_I|$ is isotone in I and hence not all possible chains start in Tss_1.) But, although $F^\omega(Ts_I) \in F^\omega(Tss_I)$, it is not necessarily the case that $F^\omega(Ts_I) \in F^\xi(Tss_I)$ because $F^\xi(Tss_I)$ is the *Prefix-reduction* of $F^\omega(Tss_I)$. However, if $F^\omega(Ts_J) \in F^\xi(Tss_J)$. then $F^\omega(Ts_{J+1}) \in F^\xi(Tss_{J+1})$. because $F^\omega(Ts_J) \in F^\xi(Tss_J)$ means that for all $F^\omega(Tsa_J) \in F^\xi(Tss_J)$: $F^\omega(Ts_J) \not\preccurlyeq F^\omega(Tsa_J)$ and $F^\omega(Tsa_J) \not\preccurlyeq F^\omega(Ts_J)$. But then $Ts_J \preccurlyeq Ts_{J+1}$ and $Tsa_J \preccurlyeq Tsa_{J+1}$ taken together imply that $F^\omega(Ts_J) \preccurlyeq F^\omega(Ts_{J+1})$ and that $F^\omega(Tsa_J) \preccurlyeq F^\omega(Tsa_{J+1})$. Thus $F^\omega(Ts_{J+1}) \not\preccurlyeq F^\omega(Tsa_{J+1})$ and $F^\omega(Tsa_{J+1}) \not\preccurlyeq F^\omega(Ts_{J+1})$ by Lemma 4.1. Therefore, every chain of $F^\omega(Ts_I)$ (an element of $F^\omega(Tss_I)$) has a closed-above subchain $F^\omega(Ts_{J\geq I})$ (an element of $F^\xi(Tss_{J\geq I})$) which is disjoint from all other such chains and thereby establishes the necessary $\omega + 1$ sequence of injections from $F^\xi(Tss_K)$ to $F^\xi(Tss_{K+1})$ and on to $F^\xi(Tss)$. This proves that $\{F^\xi(Tss_1), F^\xi(Tss_2), \ldots, F^\xi(Tss)\}$ is a chain and $F^\xi(Tss)$ is its supremum. ⊠

Theorem 6.2: The extended operator *Hold*$^\xi$ is continuous.

The continuity of *Hold*$^\xi$ follows easily from Theorem 6.1; we merely observe that "$\oplus$" is isotone on streams. Note that our results in Chapter 4 concerning streams carry over to tagged-streams, since the pairs ⟨*Datum*, *Tag-set*⟩ make perfectly good stream elements, that is, the codomain of a stream function may be any set with an equality relation. The only care we must take is to show that our resultant tagged-streams obey the tag-set extension rule. It is clear that a tagged-stream whose first element has an empty tag-set obeys the tag-set extension rule if the remainder of the tagged-stream obeys the rule, which it does by assumption, being the input. ⊠

Theorem 6.3: The extended operator *Sop*$^\xi$ is continuous in all of its arguments.

To show that *Sop*$^\xi$ is continuous on tagged-stream-sets, we first note that *Sop*$^\xi$ is an extension of *Sop*$^\omega$, which obeys the precondition of Theorem 6.1, because *Sop*$^\omega$ is the continuous completion of *Sop* (by Lemma 6.7) if *Sop* is isotone. So we need only prove that *Sop* is isotone on tagged-streams. We prove that *Sop* is isotone in its I-th (Data-path) argument by showing that if $Ts_I \leq Tsx_I$ then $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N) \preccurlyeq Sop_F(Ts_1, \ldots, Tsx_I, \ldots, Ts_N)$. The proof proceeds by induction on the finite ordinal $Dom(Tsx_1) = N$; note that $Tsx_1 = \langle\,\rangle$ iff $Dom(Tsx_1) = \{\,\}$, and that $Sop_F(Ts_1, \ldots, \langle\,\rangle, \ldots,$

$Ts_N) = \langle\,\rangle$ for any $1 \le I \le N$. Substituting Tsx_I in the definition of $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N)$, we get:

```
$Sop_F(Ts_1, Tsx_I, Ts_N) =$
  $If\ Tsx_I = \langle\,\rangle\ Then\ \langle\,\rangle$
  $If\ \exists J \ne I: Ts_J = \langle\,\rangle\ Then\ \langle\,\rangle$
  $If\ Consistent\text{-}tags(Tag\text{-}set(Ts_1{}^1), Tag\text{-}set(Tsx_I), Tag\text{-}set(Ts_N{}^1))\ Then$
    $\langle F(Datum(Ts_I{}^1), Datum(Tsx_I), Datum(Ts_N{}^1)),$
      $Merge\text{-}tags(Tag\text{-}set(Ts_1{}^1), Tag\text{-}set(Tsx_I{}^1), Tag\text{-}set(Ts_N{}^1))\rangle \oplus$
    $Sop_F(\tau\, Ts_1, \tau\, Tsx_I, \tau\, Ts_N)$
  $Otherwise\ \langle\,\rangle$
```

We assume in the steps that follow that $\forall 1 \le I \le N: Ts_I \ne \langle\,\rangle$, since otherwise $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N) = \langle\,\rangle$. The base step is: $Tsx_I = \langle\,\rangle$ implies $Ts_I = \langle\,\rangle$ so that $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N) = \langle\,\rangle = Sop_F(Ts_1, \ldots, Tsx_I, \ldots, Ts_N)$ The induction step is: let $Dom(Tsx_I) = N + 1$. If $Ts_I = \langle\,\rangle$, then $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N) = \langle\,\rangle$, which is the prefix of any tagged-stream. If $Ts_I \ne \langle\,\rangle$, then $Ts_I{}^1 = Tsx_I{}^1$ and $\tau Ts_I \le \tau Tsx_I$. Now if $Tag\text{-}set(Ts_I{}^1)$ is not consistent with some $Tag\text{-}set(Ts_J{}^1)$, then $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N) = \langle\,\rangle$, which is the prefix of any tagged-stream. Now if $Tag\text{-}set(Ts_I{}^1)$ is consistent with all $Tag\text{-}set(Ts_J{}^1)$:

```
$Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N) =$
  $\langle F(Datum(Ts_I{}^1), \ldots, Datum(Ts_I{}^1), \ldots, Datum(Ts_N{}^1)),$
    $Merge\text{-}tags(Tag\text{-}set(Ts_1{}^1), \ldots, Tag\text{-}set(Ts_I{}^1), \ldots, Tag\text{-}set(Ts_N{}^1))\rangle \oplus$
  $Sop_F(\tau\, Ts_1, \ldots, \tau\, Ts_I, \ldots, \tau\, Ts_N)$

$Sop_F(Ts_1, \ldots, Tsx_I, \ldots, Ts_N) =$
  $\langle F(Datum(Ts_I{}^1), \ldots, Datum(Ts_I{}^1), \ldots, Datum(Ts_N{}^1)),$
    $Merge\text{-}tags(Tag\text{-}set(Ts_1{}^1), \ldots, Tag\text{-}set(Ts_I{}^1), \ldots, Tag\text{-}set(Ts_N{}^1))\rangle \oplus$
    $Sop_F(\tau\, Ts_1, \ldots, \tau\, Tsx_I, \ldots, \tau\, Ts_N)$

$Sop_F(Ts_1, \ldots, Tsx_I, \ldots, Ts_N) =$
  $\langle F(Datum(Ts_I{}^1), \ldots, Datum(Ts_I{}^1), \ldots, Datum(Ts_N{}^1)),$
    $Merge\text{-}tags(Tag\text{-}set(Ts_1{}^1), \ldots, Tag\text{-}set(Ts_I{}^1), \ldots, Tag\text{-}set(Ts_N{}^1))\rangle \oplus$
    $Sop_F(\tau\, Ts_1, \ldots, \tau\, Tsx_I, \ldots, \tau\, Ts_N)$
```

Hence, by the isotonicity of "$\oplus$" we deduce that $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N) \leq Sop_F(Ts_1, \ldots, Tsx_I, \ldots, Ts_N)$ given the inductive hypothesis, that $Sop_F(\tau Ts_1, \ldots, \tau Ts_I, \ldots, \tau Ts_N) \leq Sop_F(\tau Ts_1, \ldots, \tau Tsx_I, \ldots, \tau Ts_N)$. This is the inductive hypothesis because $\tau Tsx_I \leq \tau Ts_I$ and $Dom(\tau Tsx_I) = N$.

Next we must show that the tagged-stream which results is indeed a proper one which obeys the tag-set extension rule. To do this we apply Lemma 6.4. Assume $J < K$, and let Ts^J and Ts^K be the J-th and K-th elements in the tagged-stream output of $Sop_F(Ts_1, \ldots, Ts_I, \ldots, Ts_N)$. Also assume that all its inputs Ts_I obey the tag-set extension rule. By unwinding the recursive definition of Sop_F we see that $Tag\text{-}set(Ts^J) = Merge\text{-}tags(Tag\text{-}set(Ts_1{}^J), \ldots, Tag\text{-}set(Ts_N{}^J))$ and the same for K. Thus we can easily see that Ts obeys the tag-set extension rule — refer to the above equations for $Tag\text{-}set(Ts^J)$ and $Tag\text{-}set(Ts^K)$ and note that the inputs Ts_I obey the tag-set extension rule ($Tag\text{-}set(Ts_I{}^J) \sqsubseteq Tag\text{-}set(Ts_I{}^K)$ for all I). Therefore since each application of Sop obeys the rule we conclude that Sop^ξ obeys the tag-set extension rule as well as being isotone. ⊠

Theorem 6.4: The extended operator $Oswitch^\xi$ is continuous in both of its arguments.

Again we prove this by first proving that $Oswitch$ is isotone on tagged-streams, then appealing to Lemma 6.7 and Theorem 6.1. We prove that $Oswitch_P$ is isotone in its first argument by showing that if $Tsc \leq Tscx$ then $Oswitch_P(Tsc, Tsd) \leq Oswitch_P(Tscx, Tsd)$. The proof again proceeds by induction on the finite ordinal $Dom(Tscx) = N$; note that $Tscx = \langle\,\rangle$ iff $Dom(Tscx) = \{\,\}$ and that $Oswitch_P(\langle\,\rangle, Tsd) = \langle\,\rangle = Oswitch_P(Tsc, \langle\,\rangle)$. Substituting $Tscx$ in the definition of $Oswitch_P$, we get:

```
$Oswitch_P(Tscx, Tsd) =$
    $\textit{If } Tscx = \langle\,\rangle \lor Tsd = \langle\,\rangle \textit{ Then } \langle\,\rangle$
    $\textit{If } Datum(Tscx^1) = P \land Consistent\text{-}tags(Tag\text{-}set(Tscx^1), Tag\text{-}set(Tsd^1)) \textit{ Then}$
        $\langle Datum(Tsd^1), Merge\text{-}tags(Tag\text{-}set(Tscx^1), Tag\text{-}set(Tsd^1))\rangle \oplus$
        $Oswitch_P(\tau\, Tscx, \tau\, Tsd)$
    $\textit{If } Datum(Tscx^1) \neq P \land Consistent\text{-}tags(Tag\text{-}set(Tscx^1), Tag\text{-}set(Tsd^1)) \textit{ Then}$
        $Oswitch_P(\tau\, Tscx, \tau\, Tsd)$
    $\textit{Otherwise } \langle\,\rangle$
```

We assume in the steps that follow that $Tsd \neq \langle\,\rangle$, since for any Tsc and $Tscx$, $Oswitch_p(Tsc, \langle\,\rangle) = \langle\,\rangle = Oswitch_p(Tscx, \langle\,\rangle)$. The base step is: $Tscx = \langle\,\rangle$ implies $Tsc = \langle\,\rangle$ so that $Oswitch_p(Tsc, Tsd) = Oswitch_p(\langle\,\rangle, Tsd) = Oswitch_p(Tscx, Tsd)$. The induction step is: let $Dom(Tscx) = N+1$. If $Tsc = \langle\,\rangle$, then $Oswitch_p(Tsc, Tsd) = \langle\,\rangle$, which is the prefix of any tagged-stream. If $Tsc \neq \langle\,\rangle$, then $Tsc^1 = Tscx^1$ and $\tau Tsc \leq \tau Tscx$. Now if $Tag\text{-}set(Tsc^1)$ is not consistent with $Tag\text{-}set(Tsd^1)$, then $Oswitch_p(Tsc, Tsd) = \langle\,\rangle$, which is the prefix of any tagged-stream. But if $Tag\text{-}set(Tsc^1)$ is consistent with $Tag\text{-}set(Tsd^1)$, and $Datum(Tsc^1) \neq P$ then:

$$Oswitch_p(Tsc, Tsd) = Oswitch_p(\tau\, Tsc, \tau\, Tsd)$$

$$Oswitch_p(Tscx, Tsd) = Oswitch_p(\tau\, Tscx, \tau\, Tsd)$$

Hence, since $Dom(\tau Tscx) = N$ and $\tau Tsc \leq \tau Tscx$, we may assume the induction hypothesis, that $Oswitch_p(\tau Tsc, \tau Tsd) \leq Oswitch_p(\tau Tscx, \tau Tsd)$. If however $Tag\text{-}set(Tsc^1)$ is consistent with $Tag\text{-}set(Tsd^1)$, and $Datum(Tsc^1) = P$ then:

$$\begin{aligned} &Oswitch_p(Tsc, Tsd) = \\ &\quad \langle Datum(Tsd^1), Merge\text{-}tags(Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsd^1))\rangle \oplus \\ &\qquad Oswitch_p(\tau\, Tsc, \tau\, Tsd) \end{aligned}$$

$$\begin{aligned} &Oswitch_p(Tscx, Tsd) = \\ &\quad \langle Datum(Tsd^1), Merge\text{-}tags(Tag\text{-}set(Tscx^1), Tag\text{-}set(Tsd^1))\rangle \oplus \\ &\qquad Oswitch_p(\tau\, Tscx, \tau\, Tsd) \end{aligned}$$

$$\begin{aligned} &\text{So } Oswitch_p(Tscx, Tsd) = \\ &\quad \langle Datum(Tsd^1), Merge\text{-}tags(Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsd^1))\rangle \oplus \\ &\qquad Oswitch_p(\tau\, Tscx, \tau\, Tsd) \end{aligned}$$

Hence, by isotonicity of "$\oplus$" we deduce that $Oswitch_p(Tsc, Tsd) \leq Oswitch_p(Tscx, Tsd)$ given the induction hypothesis, that $Oswitch_p(\tau Tsc, \tau Tsd) \leq Oswitch_p(\tau Tscx, \tau Tsd)$.

Now we must show that *Oswitch* obeys the tag-set extension rule. Again we apply Lemma 6.4, this time to the tagged-stream output of $Oswitch_p$. Assume $Jout \leq Kout$, and let Ts^{Jout} and Ts^{Kout} be the *Jout*-th and *Kout*-th elements of $Oswitch_p(Tsc, Tsd)$. Note however that Ts^{Kout} does not necessarily derive from Tsc^{Kout} and Tsd^{Kout} because the recursion schema skips elements of the input tagged-stream (i.e. whenever

$Datum(Tsc^L) \neq P$). But $Tag\text{-}set(Ts^{Kout}) = Merge\text{-}tags(Tag\text{-}set(Tsc^{Kin}), Tag\text{-}set(Tsd^{Kin}))$ for some $Kin \geq Kout$ and similarly for $Jout$ and Jin. Then the same argument we used to show that the output of Sop_P obeys the tag-set extension rule shows that the output of $Oswitch_p$ does. Therefore we conclude that $Oswitch_p^{\xi}$ obeys the tag-set extension rule as well as being isotone. $\boxtimes$

Theorem 6.5: The extended operator $Iswitch^{\xi}$ is continuous in all of its arguments (by Lemma 6.7 and Theorem 6.1).

The proof that *Iswitch* is isotone is more difficult. First we show that *Iswitch* is isotone on tagged-streams, then the isotonicity of $Iswitch^{\xi}$ follows directly from Theorem 6.1. We prove that *Iswitch* is isotone in all its Ts_I arguments by showing that if for all $0 \leq I \leq N$, $Ts_I \leq Tsx_I$, then $Iswitch(Tgs, Tsc, Ts_0, \ldots, Ts_N) \leq Iswitch(Tgs, Tsc, Tsx_0, \ldots, Tsx_N)$ for any $Tgs \in Tag\text{-}set$. This time the proof proceeds by simultaneous induction on the finite ordinals $Dom(Tsx_I) = N_I$; note that $Dom(Tsx_I) = \{\,\}$ iff $Tsx_I = \langle\,\rangle$, and that $Iswitch(Tgs, \langle\,\rangle, Ts_0, \ldots, Ts_N) = \langle\,\rangle$, but that it is not necessarily true that $Iswitch(Tgs, Ts_0, \ldots, \langle\,\rangle, \ldots, Ts_N) = \langle\,\rangle$. Substituting Tsx_I (for all I) into the definition of *Iswitch* we get:

```
$Iswitch(Tgs, Tsc, Tsx_0, \ldots, Tsx_N) =$
    $\textbf{If } Tsc = \langle\,\rangle \textbf{ Then } \langle\,\rangle$
    $\textbf{If } Datum(Tsc_1) = 0 \wedge Tsx_0 = \langle\,\rangle \textbf{ Then } \langle\,\rangle$
      •     •     •     •     •
    $\textbf{If } Datum(Tsc_1) = N \wedge Tsx_N = \langle\,\rangle \textbf{ Then } \langle\,\rangle$
    $\textbf{If } Datum(Tsc_1) = 0 \wedge Consistent\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_0^{\,1})) \textbf{ Then}$
      $\langle Datum(Tsx_0^{\,1}), Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_0^{\,1}))\rangle \oplus$
      $Iswitch(Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_0^{\,1})),$
                  $\tau\, Tsc, \tau\, Tsx_0, \ldots, Tsx_N)$
      •     •     •     •     •     •     •     •     •
    $\textbf{If } Datum(Tsc_1) = N \wedge Consistent\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_N^{\,1})) \textbf{ Then}$
      $\langle Datum(Tsx_N^{\,1}), Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_N^{\,1}))\rangle \oplus$
      $Iswitch(Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_N^{\,1})),$
                  $\tau\, Tsc, Tsx_0, \ldots, \tau\, Tsx_N)$
    $\textbf{Otherwise } \langle\,\rangle$
```

We assume in the steps that follow that $Tsc \neq \langle\,\rangle$, since for any Ts_I, $Iswitch(Tgs, \langle\,\rangle, Ts_0, \ldots, Ts_N) = \langle\,\rangle$. The base step is: for all $0 \leq I \leq N$, $Tsx_I = \langle\,\rangle$ which implies for all $0 \leq I \leq N$, $Ts_I = \langle\,\rangle$ so that $Iswitch(Tgs, Tsc, Ts_0, \ldots, Ts_N) =$ $Iswitch(Tgs, Tsc, \langle\,\rangle, \ldots, \langle\,\rangle) =$ $Iswitch(Tgs, Tsc, Tsx_0, \ldots, Tsx_N)$. The induction step is: let $\sum_{0 \leq J \leq N} Dom(Tsx_J) = N + 1$. There are several cases to consider depending on whether $Datum(Tsc^1) = I$ or not, whether $Ts_I = \langle\,\rangle$ or not, and whether the tag-sets are consistent or not.

If $Datum(Tsc^1) = I$ and $Ts_I = \langle\,\rangle$, then $Iswitch(Tgs, Tsc, Ts_0, \ldots, Ts_N) = \langle\,\rangle$ which is the prefix of any tagged-stream. If $Datum(Tsc^1) = I$ and $Ts_I \neq \langle\,\rangle$, then $Ts_I{}^1 = Tsx_I{}^1$ and $\tau Ts_I \leq \tau Tsx_I$. Now if Tgs, $Tag\text{-}set(Tsc^1)$ and $Tag\text{-}set(Ts_I{}^1)$ are not mutually consistent, then $Iswitch(Tgs, Tsc, Ts_0, \ldots, Ts_N) = \langle\,\rangle$, which is the prefix of any tagged-stream. But if they are mutually consistent, then:

$$
\begin{aligned}
&Iswitch(Tgs, Tsc, Ts_0, \ldots, Ts_I, \ldots, Ts_N) = \\
&\quad \langle Datum(Ts_I{}^1), Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_I{}^1))\rangle \oplus \\
&\qquad Iswitch(Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_I{}^1)), \\
&\qquad\qquad \tau\, Tsc, Ts_0, \ldots, \tau\, Ts_I, \ldots, Ts_N)
\end{aligned}
$$

$$
\begin{aligned}
&Iswitch(Tgs, Tsc, Tsx_0, \ldots, Tsx_I, \ldots, Tsx_N) = \\
&\quad \langle Datum(Tsx_I{}^1), Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_I{}^1))\rangle \oplus \\
&\qquad Iswitch(Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Tsx_I{}^1)), \\
&\qquad\qquad \tau\, Tsc, Ts_0, \ldots, \tau\, Tsx_I, \ldots, Ts_N)
\end{aligned}
$$

$$
\begin{aligned}
&Iswitch(Tgs, Tsc, Tsx_0, \ldots, Tsx_I, \ldots, Tsx_N) = \\
&\quad \langle Datum(Ts_I{}^1), Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_I{}^1))\rangle \oplus \\
&\qquad Iswitch(Merge\text{-}tags(Tgs, Tag\text{-}set(Tsc^1), Tag\text{-}set(Ts_I{}^1)), \\
&\qquad\qquad \tau\, Tsc, Ts_0, \ldots, \tau\, Tsx_I, \ldots, Ts_N)
\end{aligned}
$$

Hence, by isotonicity of "$\oplus$" we conclude that $Iswitch(Tgs, Tsc, Ts_0, \ldots, Ts_I, \ldots, Ts_N) \leq$ $Iswitch(Tgs, Tsc, Tsx_0, \ldots, Tsx_I, \ldots, Tsx_N)$ given the induction hypothesis, that $Iswitch(Tgsm, \tau Tsc, Ts_0, \ldots, \tau Ts_I, \ldots, Ts_N) \leq Iswitch(Tgsm, \tau Tsc, Ts_0, \ldots, \tau Tsx_I, \ldots, Ts_N)$ where $Tgsm = Merge\text{-}tags(\ldots)$. In any case, we are reducing $\sum_{0 \leq J \leq N} Dom(Tsx_J)$, so the induction is well founded.

Proving that Iswitch is isotone in its first argument (Tsc) is a relatively straightforward induction (similar to that of *Sop*) and is therefore omitted.

The proof that *Iswitch* obeys the tag-set extension rule is more complicated than any of the previous such proofs. The reason for this is that the recursion schema includes an extra variable *Tgs*, which accumulates the tag-sets generated by the previous recursion levels.

Now let $Ts = Iswitch(\{\ \}, Ts_0, \ldots, Ts_N)$ and consider Ts^J. Upon unwinding the recursion, we see that $Tag\text{-}set(Ts_J) = Merge\text{-}tags(Merge\text{-}tags(\ldots, Tag\text{-}set(Tsc^{J-1}), Tag\text{-}set(Ts_{Pa}{}^{Qa})), Tag\text{-}set(Tsc^J), Tag\text{-}set(Ts_{Pb}{}^{Qb}))$ (assuming Ts^J even exists). Now by associativity and commutativity of *Merge-tags*, we see that $Tag\text{-}set(Ts_J) = Merge\text{-}tags(Tag\text{-}set(Tsc^J), Tag\text{-}set(Tsc^{J-1}), \ldots, Tag\text{-}set(Ts_{Pa}{}^{Qa}), Tag\text{-}set(Ts_{Pb}{}^{Qb}), \ldots) = Merge\text{-}tags(Merge\text{-}tags(Tag\text{-}set(Tsc^J), Tag\text{-}set(Tsc^{J-1}), \ldots), Tag\text{-}set(Ts_{Pa}{}^{Qa}), Tag\text{-}set(Ts_{Pb}{}^{Qb}), \ldots)$ But by since *Tsc* obeys the tag-set extension rule, $Tag\text{-}set(Tsc^{J-1}) \sqsubseteq Tag\text{-}set(Tsc^J)$ etc., thus by Lemma 6.5 we get $Tag\text{-}set(Ts^J) = Merge\text{-}tags(Tag\text{-}set(Tsc^J), Tag\text{-}set(Ts_{Pa}{}^{Qa}), Tag\text{-}set(Ts_{Pb}{}^{Qb}), \ldots)$. Now we apply associativity and commutativity of *Merge-tags* again in order to group together the *Ts*s with the same subscript (i.e. to group together the data inputs) to get $Merge\text{-}tags(Tag\text{-}set(Tsc^J), Merge\text{-}tags(Tag\text{-}set(Ts_0{}^1), \ldots), \ldots, Merge\text{-}tags(Tag\text{-}set(Ts_N{}^1), \ldots))$. (Although we show $Ts_I{}^1$ for all I, it must be understood that the whole *M-tags* subexpression is present iff $\exists M \le J: I \in Datum(Tsc^M)$). Now by N applications of Lemma 6.5, we derive that $Tag\text{-}set(Ts_J) = Merge\text{-}tags(Tag\text{-}set(Tsc^J), Tag\text{-}set(Ts_0{}^{Count(Tsc,0,J)}), \ldots, Tag\text{-}set(Ts_N{}^{Count(Tsc,N,J)}))$ where $Count(Tsc, I, J)$ is the number of times the value I appears in the set $\{Datum(Tsc^M) \mid M \le J\}$. By the same argument, we also derive that $Tag\text{-}set(Ts_K) = Merge\text{-}tags(Tag\text{-}set(Tsc^K), Tag\text{-}set(Ts_0{}^{Count(Tsc,0,K)}), \ldots, Tag\text{-}set(Ts_N{}^{Count(Tsc,N,K)}))$ (again assuming that Ts^K even exists). Since *Tsc* and all Ts_I obey the tag-set extension rule, we see that if $K \ge J$ then $Tag\text{-}set(Tsc^J) \sqsubseteq Tag\text{-}set(Tsc^K)$ and $Tag\text{-}set(Ts_I{}^{Count(Tsc,I,J)}) \sqsubseteq Tag\text{-}set(Ts_I{}^{Count(Tsc,I,K)})$ for all $I \le N$ (since $Count(Tsc, I, J) \le Count(Tsc, I, K)$ for all $I \le N$). Thus by $N + 1$ applications of Lemma 6.4, we conclude that $Tag\text{-}set(Ts^J, Tag\text{-}set(Ts_K))$ so that *Ts* obeys the tag-set extension rule too. Therefore we have proved that $Iswitch^\xi$ obeys the tag-set extension rule as well as being isotone. ⊠

Definition of Non-determinate Primitive *Arbiter*

In order to define the *Arbiter* operator, we first define an auxiliary function *Extend-tags* which takes two arguments *Choice* and *Tgs*, and a parameter *A*. *Extend-tags* appends the number *Choice* on to the tail end of the *Choice-sequence* in the *Tag*, *Tg* (in *Tgs*), whose *Arbiter-name* is *A*. Its precise definition is:

$$\begin{aligned}&\textit{Extend-tags}_A(\textit{Choice},\textit{Tgs}) = \\ &\quad \{\, Tg \in Tgs \mid \textit{Arbiter-name}(Tg) \neq A \,\} \cup \\ &\quad \{\, (A, \textit{Choice-sequence}(Tg) \oplus \textit{Choice}) \mid Tg \in Tgs \wedge \textit{Arbiter-name}(Tg) = A \,\}\end{aligned}$$

$$\begin{aligned}&\textbf{Where } S \oplus X = Sx \\ &\quad \textbf{And } Dom(Sx) = Dom(S) + 1 \\ &\quad \textbf{And } Sx^I = \textbf{If } I \in Dom(S) \textbf{ Then } S_I \textbf{ Otherwise } X\end{aligned}$$

The $Arbiter_A$ operator takes $N+1$ arguments which are tagged-stream-sets, Tss_I, and one parameter *A* which is the *Arbiter-name* (we omit further references to *A* in the explanation). The $Arbiter_A$ applies $Arbmerge_A$ to each $N+1$-tuple from the Cartesian product of the *Tss*'s and *Prefix-reduces* the result. $Arbmerge_A$ takes $N+1$ tagged-streams and merges them all possible ways, producing a *Prefix-reduced* set of tagged-streams as its result. It does this by using $Arbmerge_{A,I}$ for each $I \leq N$ and taking the union of their results. Each $Arbmerge_{A,I}$ uses $Arbmerge_A$ recursively to merge the tail of the *I*th tagged-stream with all the rest, and attaches the head of the *I*th tagged-stream to each tagged-stream in the resulting set. The *N* sets which result at each recursion level are united to form a single set which is the overall result of that level. $Arbmerge_A$ and $Arbmerge_{A,I}$ both take an additional argument, *Tgs* (initially the set with one *Tag* whose *Arbiter-name* is *A* and whose *Choice-sequence* is empty), which records the arbitrary choices made so far in the recursion. The precise definitions of $Arbiter_A$, $Arbmerge_A$ and $Arbmerge_{A,I}$ are:

$$\begin{aligned}&Arbiter_A(Tss_0,\ldots,Tss_N) = \\ &\quad \{\, Tsa \in \bigcup Arbmerge_A{}^{\omega}(Tgs_A, Tss_0,\ldots,Tss_N) \mid \\ &\qquad \forall Tsb \in \bigcup Arbmerge_A{}^{\omega}(Tgs_A, Tss_0,\ldots,Tss_N) : Tsa \not< Tsb \,\}\end{aligned}$$

$$\textit{Where } Tgs_A = \{(A, \langle\,\rangle)\}$$

$$\begin{aligned}&\textit{And } Arbmerge_A(Tgs, Ts_0,\ldots,Ts_N) = \\ &\quad \{\, Tsa \in \bigcup_{I \leq N} Arbmerge_{A,I}(Tgs, Ts_0,\ldots,Ts_N) \mid \\ &\qquad \forall Tsb \in \bigcup_{I \leq N} Arbmerge_{A,I}(Tgs, Ts_0,\ldots,Ts_N) : Tsa \not< Tsb \,\}\end{aligned}$$

```
And $Arbmerge_{A,I}(Tgs, Ts_0, \ldots, Ts_N) =$
  If $Ts_I = \langle\rangle$ Then $\{\langle\rangle\}$
  If $Consistent\text{-}tags(Extend\text{-}tags_A(I, Tgs), Tag\text{-}set(Ts_I^1))$ Then
    $\{\langle Datum(Ts_I^1), Tgsx\rangle \oplus Ts \mid$
      $Ts \in Arbmerge_A(Tgsx, Ts_0, \ldots, \tau\, Ts_I, \ldots, Ts_N) \wedge$
      $Tgsx = Merge\text{-}tags(Extend\text{-}tags_A(I, Tgs), Tag\text{-}set(Ts_I^1))\}$
  Otherwise $\{\langle\rangle\}$
```

The fact that $Arbmerge_A^{\omega}$ is continuous on its tagged-stream arguments, even though its result is a tagged-stream-set, will be made clear by Lemma 6.10. Note that the uses of $Arbmerge_A^{\omega}$ in the definition of $Arbiter_A$ make use of our function-of-argument-sets convention only with respect to the Tss_I: although Tgs_A is a set, $Arbmerge_A^{\omega}$ wants a such a set as its first argument.

Continuity of the Non-determinate *Arbiter*

To prove that $Arbiter_A$ is isotone, we first prove that $Arbmerge_A$ is isotone in its tagged-stream arguments. Note that the output domain is the domain of *Tagged-stream-sets* while the input domain is that of *Tagged-streams*. Since they are both posets however, isotonicity is well defined. First, however, we prove another handy lemma about tag-sets.

Lemma 6.8: For any legitimate tag-set Tgs, $Tgs \sqsubseteq Extend\text{-}tags_A(C, Tgs)$.

From the definition of *Extend-tags* we see that an element Tgx of $Extend\text{-}tags_A(C, Tgs)$ is either already an element of Tgs (if $Arbiter\text{-}name(Tgx) \neq A$) or it derives from the element Tga in Tgs such that $Arbiter\text{-}name(Tga) = Arbiter\text{-}name(Tgx)$ and $Choice\text{-}sequence(Tg) \leq Choice\text{-}sequence(Tgx)$ (if $Arbiter\text{-}name(Tgx) = A$). ⊠

Lemma 6.9: If $\forall I \leq N: Ts_I \leq Tsx_I$ then $Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N) \leq Arbmerge_A(Tgs, Tsx_0, \ldots, Tsx_N)$.

To show that this is true, we need an injection from $Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N)$ to $Arbmerge_A(Tgs, Tsx_0, \ldots, Tsx_N)$ such that each element of the first is a prefix ("$\leq$") of an element of the second. Since $Ts_I \leq Tsx_I$ for all $I \leq N$, we see that $Tsx_I^1 = Ts_I^1$

(assuming the non-trivial case $Dom(Ts_I) \neq \{\ \}$). Upon substituting Tsx_I for Ts_I in the definitions of $Arbmerge_A$ and $Arbmerge_{A,I}$ we get:

```
$Arbmerge_A(Tgs, Tsx_0, \ldots, Tsx_N) =$
  $\{ Tsa \in \bigcup_{I \le N} Arbmerge_{A,I}(Tgs, Tsx_0, \ldots, Tsx_N) \mid$
    $\forall Tsb \in \bigcup_{I \le N} Arbmerge_{A,I}(Tgs, Tsx_0, \ldots, Tsx_N) : Tsa \not< Tsb \}$

$And\ Arbmerge_{A,I}(Tgs, Tsx_0, \ldots, Tsx_N) =$
  $If\ Tsx_I = \langle\,\rangle\ Then\ \{\langle\,\rangle\}$
  $If\ Consistent\text{-}tags(Extend\text{-}tags_A(I, Tgs), Tag\text{-}set(Ts_I{}^1))\ Then$
    $\{ \langle Datum(Ts_I{}^1), Tgsx \rangle \oplus Tsx \mid$
      $Tsx \in Arbmerge_A(Tgsx, Tsx_0, \ldots, \tau\ Tsx_I, \ldots, Tsx_N) \wedge$
      $Tgsx = Merge\text{-}tags(Extend\text{-}tags_A(I, Tgs), Tag\text{-}set(Ts_I{}^1)) \}$
  $Otherwise\ \{\langle\,\rangle\}$
```

Now consider a stream Tsa in $Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N)$, and consider its first element Tsa^1. Clearly, $Tsa \in Arbmerge_{A,I}(Tgs, Ts_0, \ldots, Ts_N)$ for some I $Tsa^1 = \langle Datum(Ts_I{}^1), Tgsx \rangle$, and $\tau Tsa \in Arbmerge_A(Tgsx, Ts_0, \ldots, \tau Ts_I, \ldots, Ts_N)$, where $Tgsx = Merge\text{-}tags(Extend\text{-}tags_A(I, Tgs), Tag\text{-}set(Ts_I{}^1))$. From this we conclude that Tsa may be characterized by its decision sequence or "oracle" J_K where $K \in Dom(Tsa)$. The same oracle may be applied to the elaboration of:

```
$Arbmerge_A(Tgs, Tsx_0, \ldots, Tsx_N) =$
  $\{ Tsa \in \bigcup_{I \le N} Arbmerge_{A,I}(Tgs, Tsx_0, \ldots, Tsx_N) \mid$
    $\forall Tsb \in \bigcup_{I \le N} Arbmerge_{A,I}(Tgs, Tsx_0, \ldots, Tsx_N) : Tsa \not< Tsb \}$
```

This will give rise to a set of one or more streams since the recursion can proceed at least as far as before (because $Ts_I \le Tsx_I$ for all I). If we pick an arbitrary stream $Tsax$ from this set, we easily see that $Tsa \le Tsax$ since the oracle J_K that generates Tsa generates a prefix of $Tsax$, and the elements of that prefix are equal to the corresponding elements of Tsa since those elements derive from the Ts_I prefixes of the Tsx_I. Furthermore, if $Tsb \neq Tsa$ is in $Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N)$ the $Tsbx$ in $Arbmerge_A(Tgs, Tsx_0, \ldots, Tsx_N)$ of which it is a prefix must not equal $Tsax$ by Lemma 4.1. Thus we have established an injection from $Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N)$ to $Arbmerge_A(Tgs, Tsx_0, \ldots, Tsx_N)$ such that each element of the domain is a prefix of its image. Therefore, $Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N) \le Arbmerge_A(Tgs, Tsx_0, \ldots, Tsx_N)$.

Each stream in $Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N)$ obeys the tag-set extension rule. This follows easily from Lemmas 6.6 and 6.8.

The result of all this is that $Arbmerge_A$ is indeed isotone in its tagged-stream arguments and each element of its output prefix-reduced tagged-stream-set is a proper tagged-stream. ⊠

Lemma 6.10: The completion $Arbmerge_A{}^{\omega}$ of $Arbmerge_A$ is continuous in its tagged-stream arguments Ts_I.

The fact that the output of $Arbmerge_A$ is a tagged-stream-set does not upset the continuity result of Lemma 6.7. All that is required is that the codomain is ω directed set complete, which it is since it is ω chain complete.

Now we can state and prove the key result of this chapter: the theorem that completes the basis for a denotational semantics of non-determinate data flow programs. It is principally for this theorem that the chain complete poset of ***Tagged-stream-sets*** was developed in the last chapter.

Theorem 6.6: The non-determinate *Arbiter* operator is continuous in each of its (tagged-stream-set) arguments.

The proof of this is similar to the proof of Theorem 6.1. First, to shorten the text of the proof, we abbreviate $Arbmerge_A(Ts_0, \ldots, Ts_P, \ldots, Ts_N)$ as $Amp(Ts_P)$ and $Arbiter_A(Tss_0, \ldots, Tss_P, \ldots, Tss_N)$ as $Ap(Tss_P)$, where the P-th argument is the one of interest. (During the rest of the proof, Ts_I and Tss_I will refer to an element of the respective chain, not to the I-th argument of $Arbmerge_A$ or $Arbiter_A$, unless otherwise stated or implied by appearance as an explicit argument.) Now, by Lemma 6.10 we know that Amp^{ω} is continuous and thus isotone. Let $\{Tss_1, Tss_2, \ldots, Tss\}$ be a chain whose supremum is Tss, and consider the sequence of image sets $\{Amp^{\omega}(Tss_1), Amp^{\omega}(Tss_2), \ldots, Amp^{\omega}(Tss)\}$. Note that this is not the extension of Amp^{ω}, just the application of a function to a set of arguments. Now if $\{Ts_K, Ts_{K+1}, \ldots, Ts\}$ (where $Ts_I \in Tss_I$) is a chain whose supremum is Ts, then $\{Amp^{\omega}(Ts_K), Amp^{\omega}(Ts_{K+1}), \ldots, Amp^{\omega}(Ts)\}$ is a chain whose supremum is $Amp^{\omega}(Ts)$, by Lemma 6.10. We wish to establish an $\omega + 1$ sequence of injections from $Ap(Tss_K)$ to $Ap(Tss_{K+1})$ and on to $Ap(Tss)$ which demonstrates that they form a chain. Here we must depart from the proof of Theorem 6.1 since the result of $Amp(Ts)$ is a tagged-stream-set rather than a tagged-stream. Noting that $Arbiter_A(\{Ts_0\}, \ldots, \{Ts_N\}) = Arbmerge_A(Tgs, Ts_0, \ldots, Ts_N)$, and that $Ap(Tssa) \leq Amp^{\omega}(Tssa)$, suggests that we form the union of the maps which show that $Amp^{\omega}(Ts_K) \trianglelefteq Amp^{\omega}(Ts_{K+1})$, uniting over all $Ts_K \in Tss_K$. We do this and then restrict the domain of this relation to the set $Ap(Tss_K)$ to get a function F_K, since we thereby discard any first elements of the relation pair which were equal. That this function is injective follows easily from Lemma 4.1. Suppose $F_K(Tsa) = F_K(Tsb)$ where $Tsa, Tsb \in Ap(Tss_K)$. Then either $Tsa \leq Tsb$ or $Tsb \leq Tsa$ by Lemma 4.1. But $Ap(Tss_K)$ is prefix-reduced, so this is impossible. Hence F_K is an injection from $Ap(Tss_K)$ to $Ap(Tss_{K+1})$ such that each element is a prefix of its image, which establishes that $Ap(Tss_K) \trianglelefteq Ap(Tss_{K+1})$. By repeating this construction sufficiently often ($\omega + 1$ times!) we establish that $\{Tss_1, Tss_2, \ldots, Tss\}$ is indeed a chain of tagged-stream-sets whose supremum is Tss, since each element of Tss is the supremum of a chain of tagged-streams. Therefore we have proved that Ap is continuous, but P was arbitrary, so $Arbiter_A$ is continuous in each argument. $\boxtimes$

First Order Fix-Points of Non-determinate DFPL Programs

Having established that all the DFPL primitive operators are continuous in their tagged-stream-set arguments, we conclude that any recursion-free DFPL program can be solved for its first order fixed-point behavior. as indicated in Chapter 4. The details of the data domain do not matter, as long as it is ω-chain complete. Similarly, the details of the operators do not matter, as long as they are continuous functions on the domains. We will therefore undertake a simple fixed-point computation.

Figure 6.1 shows a non-determinate DFPL program with a loop, for which we will compute a first order fixed-point. Note that this time we introduce input from outside the loop. We do this to get a non-trivial answer since none of the operators in the loop generate any data. The *Every-other* operator is as defined in Chapter 4 and will again ensure finiteness of the result. The *Arbiter* operator is as defined earlier in this chapter except that we drop the parameter A which distinguishes among *Arbiter*'s since we only have one of them.

To solve this loop, we cut it at the point labeled X, then we solve the equation $X = Arbiter(\{\langle A, B\rangle\}, Every\text{-}other(X))$. To make the solution process more illuminating, we introduce the auxiliary variable Y, and generate approximate solutions to the above equation in two steps: $Y_I = Every\text{-}other(X_I)$ and $X_{I+1} = Arbiter(\{\langle A, B\rangle\}, Y_I)$. Naturally we start the approximation with $X_0 = \bot = \{\langle\rangle\}$. The first approximation is:

$$X_1 = Arbiter(\{\langle A, B\rangle\}, \{\langle\rangle\}) = \{\langle A_0, B_{00}\rangle\}$$

$$Y_1 = Every\text{-}other(X_1) = \{\langle A_0\rangle\}$$

The second approximation is:

$$X_2 = Arbiter(\{\langle A, B\rangle\}, Y_1) = \{\langle A_0, B_{00}, A_{001}\rangle, \langle A_0, A_{01}, B_{010}\rangle\}$$

$$Y_2 = Every\text{-}other(X_2) = \{\langle A_0, A_{001}\rangle, \langle A_0, B_{010}\rangle\}$$

Note that the fact that the second input to *Arbiter* is already tagged (by this selfsame *Arbiter*) constrains the way that *Arbmerge* can do its merging — the tag generated by *Extend-tags* must be consistent with the input tag. The third approximation is:

$$X_3 = Arbiter(\{\langle A, B\rangle\}, Y_2) = \{\langle A_0, B_{00}, A_{001}, A_{0011}\rangle, \langle A_0, A_{01}, B_{010}, B_{0101}\rangle\}$$

$$Y_3 = Every\text{-}other(X_3) = \{\langle A_0, A_{001}\rangle, \langle A_0, B_{010}\rangle\}$$

Note that the tags as well as the data is dropped by *Every-other* from the output tagged-streams — the output therefore indicates *only* those arbitrary decisions that actually entered into the particular output. Note also that the generation of X_3 involves *Prefix-reduction*. Part of the computation of X_3 involves evaluating $Arbmerge(\langle A, B\rangle, \langle A_0, A_{001}\rangle)$. This generates the empty tagged-stream and the tagged-stream $\langle A_0, A_{01}, B_{010}\rangle$, both of which are discarded by the *Prefix-reduction* which occurs when *Arbiter* generates its result tagged-stream-set.

Since $Y_3 = Y_2$ the fixed-point computation has converged and the solution is $X = \{\langle A_0, B_{00}, A_{001}, A_{0011}\rangle, \langle A_0, A_{01}, B_{010}, B_{0101}\rangle\}$. If we were cut the graph at Y instead of X, the first approximation would start with $Y_0 = \{\langle\rangle\}$, so that $X_1 = Every\text{-}other(Y_0) = \{\langle\rangle\}$. This would delay the convergence by 1 step, but the fixed-point would obviously be the same.

-7-
Conclusion

Overview

This chapter ties up loose ends and suggests directions for future work in the semantics of Data Flow Languages. Some of the loose ends considered are: "fairness" of non-determinate DFPL programs, functional behavior of DFPL programs with loops and recursion, and the meaning of "bottom" (or $\perp$) in DFPL's semantics. Directions for future work are suggested in the areas of: our semantics as a means to proving equivalence of DFPL programs, operators as valid DFPL data and the relation to reflexive domains.

Explanation of the Anomaly of Brock and Ackerman

In [B&A-77], Brock and Ackerman present two small non-determinate data flow programs which exhibit anomalous behavior. The anomaly is that their operational behavior is different from the behavior predicted by a simple denotational semantics based on sets of streams. From this, they correctly conclude that a semantics based only on sets of streams (which they call *histories*) is inadequate to characterize non-determinate systems. In our model, based on set of tagged streams, their two programs correspond to *different* functions, and thus their different behavior is not anomalous. In particular, the first stream element output by the second program is tagged, while the first stream element output by the first program is *not* tagged. The details of this may easily be filled in by examining their note, and will not be elaborated here.

"Fairness" and the *Arbiter*

As mentioned in Chapter 1, a non-determinate service program may or may not treat its users "fairly". The usual definition of a "fair" program is that the program never keeps the user who requests service waiting for more than a specified or reasonable period of time. This can be refined by specifying what period of time is permissible. Two possibilities are: no user need wait an infinite amount of time for a request to be serviced; no user need wait more than a bounded amount of time for a request to be serviced after it is presented to the system.

Neither of these definitions of "fairness" are directly applicable to DFPL since its semantics has no notion of time in the usual sense. The semantics of DFPL does, however, have the notion of relative order of appearance of data items. (This ordering is induced by the ordering of the positive integers which is the domain of the function which defines a *Tagged-stream*.) Thus the above definitions of "fairness" can be recast as follows: any user's request will be serviced after a finite number of other users' requests are serviced; any user's request will be serviced after a bounded number of other users' requests are serviced. Let us now investigate whether either of these definitions of "fairness" can be satisfied within the semantics of DFPL.

Since the source of all non-determinate behavior in DFPL is the *Arbiter*, the question boils down to the "fairness" of the *Arbiter*. That is, if several sources of requests are to be merged into one for consideration by some processing program, even if the program has internal queues for unsatisfied requests, the *Arbiter* makes the initial decision as to which request gets served or even queued for service.

Recall that neither in the determinate semantics of streams (cf. Chapter 4) nor in the non-determinate semantics of tagged-stream-sets (cf. Chapter 6) do we have the notion of a datum in one stream preceding or following a datum in another stream. Therefore, we cannot even express the concept of a datum not being delayed at an *Arbiter* while more than a bounded number of other inputs are processed. There is another related concept which is expressible however. That is the idea any stream which is the output of an *Arbiter* will never have more than a bounded number of contiguous data items which are passed through from any single input stream. This could easily be realized by changing the definition of the $Arbmerge_A$ subfunction (cf. Chapter 6) to have extra arguments which counted how many data items from each input have been accepted so far and constraining thereby which $Arbmerge_A^I$ subfunction was to be called at each recursion. Unfortunately, this approach has a crippling flaw. Suppose the bound on the number of contiguous acceptances is N and suppose that one input to the *Arbiter* is presented with a stream of length $N+M$ and the other inputs with empty streams. Then the output can only consist of the first N data items of the non-empty input stream, the remaining M items will *never* be accepted. Thus in the name of bounded delay "fairness", we impose *infinite* delay in certain circumstances! Therefore we can reject bounded delay "fairness" as incompatible with our fairly simple semantics of DFPL.

The notion of finite delay "fairness" still requires consideration. An *Arbiter* may be said to have finite delay if any input datum eventually shows up in the output stream no matter what sequence of arbitrary (but allowable) decisions were made by the *Arbiter*. More precisely, we would say that for any input stream Ts_I and for any J in its domain, then for any output stream Ts in the output tagged-stream-set Tss there exists a finite K such that $Ts^K = Ts_I{}^J$. Now imagine a two input *Arbiter* such that its left input is presented with the singleton stream $\langle A \rangle$ (for simplicity, we ignore the fact that the inputs are really sets), and its right input is presented with the stream $\langle B, B, \ldots, B \rangle$. The output of this *Arbiter*, according to its definition in Chapter 6, must be the tagged-stream-set $\{\langle A, B, \ldots, B \rangle, \langle B, A, \ldots, B \rangle, \ldots, \langle B, \ldots, B, A \rangle\}$. (Here we drop the tags in the interest of brevity, they are deducible from the data which are distinct.) The last stream in this tagged-stream-set has the A all the way at its end. No matter how long a finite stream of B's we feed it, the *Arbiter* will always produce such a stream as an element of its output tagged-stream-set. Since the *Arbiter* is continuous, its output when confronted with an infinite input stream is the supremum of its outputs generated from the finite input streams which have that infinite stream as their supremum.

To compute this supremum, let us adopt a bit of notation which departs somewhat from our previous notation. Let $\langle A^N \rangle$ stand for a stream of N occurrences of A, rather than the N-th element of a stream A. (We can distinguish this usage by the precise typography of the letters A, B etc.) Then our example may be written as:

$$\begin{aligned} &\mathit{Arbiter}(\langle A \rangle, \langle B^N \rangle) = \\ &\quad \{\langle B^K, A, B^{N-K} \rangle \mid 0 \le K \le N\} = \\ &\quad \{\langle B^N, A \rangle\} \cup \{\langle B^K, A, B^{N-K} \rangle \mid 0 \le K < N\} \end{aligned}$$

Now these tagged-stream-sets clearly comprise a chain for increasing N (they must because *Arbiter* is isotone), and that chain has a supremum. By our construction of such a supremum (cf. Chapter 5), each tagged-stream in each tagged-stream-set must be in a chain of tagged-streams which has a tagged-stream supremum. The question now is, does the infinite stream (B^{ω}) appear in the supremal output set? The answer is no, because there is no chain of sequences in the chain of sets which has that infinite stream as its supremum. A set does exist which is an upper bound of the chain of sets and which contains that stream, but it would not be the *least* upper bound. $\boxtimes$

Therefore, although the continuity of the *Arbiter* precludes its being fair in the bounded delay sense, it is fair in the sense of having only finite delay.

Second Order Theory

The set of continuous functions from a chain complete poset to a chain complete poset themselves form a chain complete poset under the pointwise order [Mar-77, Ros-77]. That is, $F \sqsubseteq G$ iff $\forall X: F(X) \leq G(X)$. Thus, equations in such functions also have fixed-points which can, in principle, be computed by the same method. We call the theory of fixed-points in the function domain the second order theory to distinguish it from the first order theory of fixed-points in the domain of streams. There are two basic classes of DFPL programs whose functional fixed-points are of interest: the iterative and the recursive. We shall first consider these separately in order to see how programs which have both iterative and recursive parts may be dealt with.

Figure 7.1 shows a prototypical DFPL defined operator with an iterative body. We wish to determine what F is given G. That is, we have the equation $Y = G(Y, X)$ but we desire an F such that $Y = F(X)$. This schema is sufficiently general to encompass all iterative procedures. The variables X and Y may in general be tuples of tagged-stream-sets where Y includes all feedback and output paths and G includes the entire body of the procedure. If not all feedback paths are desired as outputs, an appropriate projection function can be applied to Y to yield the outputs, but this changes the solution in a trivial way only.

Let us assume that G is continuous and that $G: D \times D \rightarrow D$ where D is our chain complete domain. Let $G_X = \lambda Y . G(X, Y)$. Then G_X is continuous in its single argument. To solve $Y = G(X, Y)$ for a given X is equivalent to solving $Y = G_X(Y)$, which is done by finding $\bigsqcup\{\perp, G_X(\perp), G_X(G_X(\perp))\}$. To solve for the F above however, we must allow X to be an actual argument. Therefore consider the sequence:

$$F_0 = \lambda X . \perp$$

$$F_1 = \lambda X . G(X, \perp)$$

$$F_2 = \lambda X . G(X, G(X, \perp))$$

$$F_3 = \lambda X . G(X, G(X, G(X, \perp)))$$

$$\bullet \quad \bullet \quad \bullet \quad \bullet$$

Clearly each F_I is a continuous function, because G is continuous in each argument, and both partial application and composition preserve continuity. Also, the set $\{F_I \mid I \geq 0\}$ is a chain in the function domain, that is, $F_0 \leq F_1 \leq F_2 \leq \ldots$ This follows by induction from the facts that G is isotone in its second argument and $\forall X: \perp \leq G(X, \perp)$. Therefore, the chain has a continuous supremum $F = \bigsqcup\{F_I \mid I \geq 0\}$ which is the continuous function we wanted.

We conclude from this argument that any DFPL procedure which has a loop for a body has a well defined semantic function which describes its behavior.

Second Order Fixed Points of Recursive Programs

Figure 7.2 shows a prototypical DFPL defined operator with an recursive body. Again, by suitable bundling of data paths and repackaging of operators, any recursively defined operator can be made to look like F. The equation to be solved is thus $Y = F(X) = H(Gl(X), F(Gr(X)))$, where Gl is the part of G that generates the left output and Gr is the part that generates the right output. Since this equation must hold for all X, we abstract to get $F = \lambda X . H(Gl(X), F(Gr(X)))$. Abstracting once again, we convert this to the second order (or functional) equation $F = \lambda E . \lambda X . H(Gl(X), E(Gr(X)))(F)$. Thus we wish to find the (second order) fixed-point of the functional $\lambda E . \lambda X . H(Gl(X), E(Gr(X)))$. But we know that any such functional, consisting of compositions of (first order) continuous functions and function variables, is (second order) continuous. Therefore, it has a least fixed-point, and that fixed-point is the recursively defined function F.

We deduce from this argument that any DFPL procedure which has a recursive body corresponds to a well defined semantic function which describes its behavior. Furthermore, DFPL procedures which involve both loops and recursion can be solved in a similar manner to yield their overall semantic function. The proper way of determining the semantic function of a large program, consisting of many procedure definitions and uses, is of course to solve for the semantic function of as small units as possible, then to build up the function for the whole program out of these units.

"Bottom", Strictness and Termination

In most treatments of denotational semantics [Man-74, S&S-71, Sto-76] the bottom element of the data domain represents the totally undefined datum, while the bottom element of a function domain represents the totally undefined function. The bottom datum, $\perp$, is then (reasonably enough) taken to represent the "result" of a non-terminating computation. That is, the partial function which the program computes is extended to a total function by defining it to yield $\perp$ where it was otherwise undefined. Given this interpretation of $\perp$, it is also reasonable to demand that most functions be "strict", that is, that they yield $\perp$ as their result if any of their arguments are $\perp$. This follows from the operationally reasonable notion that it is impossible to invoke a subroutine until the computations of all of its arguments are finished. Strictness is not demanded of all functions however, the *If-then-else* function is usually only strict in its predicate so that it can be used to terminate recursions and iterations.

In the semantics of DFPL, the bottom element of the function domain indeed represents the totally undefined element, but the interpretation of $\perp$ in the data domain must be different. The data domain of DFPL, recall, is based on the notion of *Streams*, therefore its bottom element is the (set consisting of) the empty stream. The empty stream, however, is definitely not the "result" of a non-terminating computation, but rather is the definite result of a computation which has not yet received enough input to generate output on that port. (Note that much output may have appeared on another port, a luxury not permitted in most programming languages.) Furthermore, DFPL functions need not be strict at all. In fact, of the primitives, only the *Oswitch* and the *Pcf*'s are strict; the *Iswitch*, *Arbiter* and especially the *Hold* (which has only one input) are not strict. Therefore, $\perp$ in the data domains of DFPL is not an "undefined" element which is added out of mathematical necessity (ie. to totalize partial functions and to clean up the partial order) but is a rather natural object which is as much a part of the notion of streams as zero is of the integers.

Program Correctness and Equivalence

As we stated in the introduction to this thesis, it is necessary to have a precise semantics for a programming language in order to be able to prove things about programs. Given a denotational semantics for a programming language, as we have for DFPL, it may be possible to determine the overall function computed by a program in that language. Having done so, it may then be possible to show that this function meets a specification (program correctness), or that it is the same overall function as that computed by another program (program equivalence). Our semantics gives a basis for doing such proofs, but it does not make them universally trivial, no interesting and useful semantics can do that. However, our semantics incorporates a model of non-determinate behavior, which many other semantics have trouble dealing with.

Referring back to Figures 2.10 and 2.11, we can now see that these two miniature programs are indeed equivalent in their overall functionality up to homomorphism (assuming that F is determinate and history independent, *i.e.* $X^I = F(U^I)$). That is, the X, Y and Z outputs of program 2.10 are (singleton) sets of tagged streams, whereas the outputs of program 2.11 are just streams, so we must map each singleton set to its element and remove all tags. This result follows directly from the denotational definitions of the various operators, we will not give the details as they consist merely of substitution into the defining equations, and then applying the homomorphism. Note that the *Arbiter* in Figure 2.10 is an augmented operator: its horizontal output is a (set of) stream(s) of index numbers, appearing in synchronization with the regular (vertical) output, such that each number merely says which input port is currently selected.

Operator Valued Data and Reflexive Domains

As mentioned in chapter 3, our current denotational semantics for DFPL does not include operators (functions) as data, and thus has no need for reflexive domains. To meaningfully add operators to DFPL as data values, we would need an *Apply* operator which would take such data as input. Unfortunately, it is not clear how to define an *Apply* operator in an informal operation sense, much less in a precise denotational sense. It is reasonable to assume that the *Apply* operator would accept a directed graph or equivalent representation of the function it was to apply, and that it would "connect" that function to the other inputs and outputs of the *Apply*, and then "start up" the new subnetwork. The problem with this model is: when does the application terminate? There is, of course, no problem with an operator which does not terminate, the peculiarity of *Apply* is that it seems inherently to never accept more than one function datum from it ostensible stream of function data. This is due to the fact that it is in general undecidable whether a subnetwork has terminated or not. One might get around this difficulty by defining a more subtle *Apply* operator.

One possibility is the following: the *Apply* operator receives as input a representation of a *running* subnetwork rather than just its function. That is, it receives the network representing the operator together with any streams in progress. It also has an input, besides the inputs and outputs which are connected to the application, which must be "pulsed" in order to make the applied subnetwork execute one transition (we are taking in operational terms again). The *Apply* operator continues running the subnetwork being applied as long as this input is pulsed with *True*. When a *False* is supplied instead, the current "state" of the subnetwork is dumped out on an auxiliary output port of the *Apply*. This output value may be fed in to the *Apply* later to resume execution. By this means we finesse the undecidability of termination of the applied subnetwork, we leave it to the user of the *Apply* to determine when to stop. This makes meaningful the notion of a stream of things to be applied; it is similar to a stream of jobs to a batch operating system.

Although, from the operational point of view, the applicable object is a function network plus its internal state, it is just another function of streams to streams from the denotational point of view, since such functions already exhibit the behavior of having an internal state. The extension to non-determinate functions should fit into this framework as it did before the *Apply* operator. This suggests that we might want a *reflexive domain* [Sto-77] as our underlying domain, that is, a domain which not only contains ordinary data but also the continuous functions from that domain to itself. This domain D would have to satisfy the equation $D \cong$ *Sets-of-tagged-streams-of*$(Q \oplus [D \to D])$, where Q is the domain of simple data (numbers, strings, records etc.), and "$\cong$" denotes isomorphism, "$\oplus$" denotes disjoint union, and $[D \to D]$ denotes the continuous functions from D to D. This is neither the usual reflexive domain equation, due to the presence of the set constructor, nor is it quite the *power domain* equation, due to the fact that the *sets of tagged streams* are not plain sets.

This needs further research, both to investigate the utility and cleanliness of this solution and to formulate it precisely in the denotational model. (Note that the *Apply* operator is not necessary in order to have the analog of subroutines, it more is like the "program loader" of conventional operating systems.)

Relation to the Lattice Formulation of Data Types

Dana Scott has developed a rather complete theory of computation based on complete lattices and continuous functions [Sco-76]. His underlying approach is to model everything in terms of one universal domain, the domain $\mathscr{P}\omega$ of all subsets of the non-negative integers, which is an *algebraic* and *continuous* lattice as well as being a topological space. In this domain, continuous functions on the domain may be represented by encodings of their *graphs* (sets of argument-value pairs), as can data values themselves. Reminiscent of Gödel numbering, encodings are sets of integers, that is, elements of $\mathscr{P}\omega$. A single number is encoded as the singleton set containing that number. The finite subsets E_N of $\mathscr{P}\omega$ may be enumerated (as $E_N = \{K_0, \dots, K_{M-1}\}$ where $N = \sum_{i<M} 2^{K_i}$) and the result of applying a (continuous) function F to an arbitrary element of the domain is defined by $F(X) = \{F(E_N) \mid E_N \subseteq X\}$. Since functions map elements of $\mathscr{P}\omega$ to elements of $\mathscr{P}\omega$, functions may have arguments and values which are sets, e.g. $6 \cup 10 + 1 = 7 \cup 11$. In fact, Scott is able to express the both the lambda calculus and a good amount of recursive function theory in this domain, including proofs of validity of lambda conversion rules, the continuity of lambda definable functions, the first and second recursion theorems, and the recursive non-enumerability of equations in the lambda calculus.

The fact that functions take sets as arguments and deliver results which are sets suggests that the domain $\mathcal{P}\omega$ might be useful for expressing non-determinacy. However, the main purpose to which Scott puts this capability is the definition of data types as virtually arbitrary subsets of $\mathcal{P}\omega$. He does this by introducing a class of functions called *retracts* which are idempotent functions on $\mathcal{P}\omega$ that map the data type to identically to itself, and other elements onto the data type. Then, by defining operators which allow combination of retracts, he is able to show that certain recursively defined data types are the minimal fixed-point solutions to equations involving retracts. For example, the data type of trees, both finite and infinite, is the solution to the equation $Tree \cong Nil + (Tree \times Tree)$ where "+" and "×" are operators on retracts analogous to union and cartesian product.

The generality of the domain $\mathcal{P}\omega$, in particular its ability to express sets of data, would make it a possibility as an underlying model of the semantics of non-determinate DFPL. Certainly the tagged-streams needed could be encoded as sets of integers as easily as functions can be. However, whether this would clarify the semantics is doubtful: encodings of this sort are rarely noted for their transparency. Nor is the explicit machinery for dealing with non-determinacy already developed in this model. The existence, completeness and continuity of the relevant domains and functions has already been established for non-determinate DFPL. The treatment of operators as data is probably better examined in the framework of *power domains* as noted below.

Relation to Power Domains

The *powerdomain* construction of Plotkin [Plo-76], as clarified by Smyth [Smy-78], bears some similarity to our poset of tagged-stream-sets, there are some important differences however. The most important is that Smyth assumes different domains, a domain S of *states*, and a domain R of *resumptions* (similar to *continuations*). The states are the states of the abstract machine, while the resumptions are mappings from states into *sets* of states (disjointly united with state, resumption pairs). The reflexive domain equation is thus $R \cong [S \rightarrow \mathcal{P}(S \oplus S \times R)]$, where the powerset constructor, $\mathcal{P}$, is needed in order to express possible non-determinacy. The problem then is, how does one solve such equations?

To do this, Smyth introduces quasi-ordered *predomains* which are sets of outcomes of (non-determinate) computations. These are ordered by the "Milner ordering", which is only a quasi-order:

$$
\begin{aligned}
&S \sqsubseteq T \equiv \\
&\quad \forall X \in S: \exists Y \in T: Y \leq X \wedge \\
&\qquad \forall Y \in T: \exists X \in S: Y \leq X
\end{aligned}
$$

The elements of this domain may be viewed as cross sections of a tree which represents the non-determinate computation; each path from the root to a leaf corresponds to a particular sequence of arbitrary choices made by an instance of the computation. Smyth observes that this model and this (quasi) ordering forces one to make "unwelcome identifications" of outcomes in forming the equivalence classes necessary for a true poset. He suggests that this could be remedied by taking arcs of the trees rather than just their cross sections. This is essentially equivalent to our use of tagged-stream-sets, except that we have streams of data rather than successive outcomes obtained by letting the computation run ever longer. Furthermore, although Smyth suggests category theory as a basis for the improved analysis, we make do with the more conventional mathematics of sets and sequences.

Since our underlying domain, tagged-stream-sets, is a true poset and has a simple structure, it seems likely that our recursive domain equation stated above can be solved in a straightforward manner using the techniques of Smyth and Plotkin. This is an especially promising area for future research.

References

A & W-77 E.A. Ashcroft and W.W. Wadge, *LUCID, a Nonprocedural Language with Iteration*, Communications of the ACM, Vol. 20, No. 7, July 1977.

ADJ-77 J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, Journal of the ACM, Vol. 24, No. 1, January 1977.

Ada-68 D.A. Adams, *A Computational Model with Data Sequenced Control*, Computation Group Tech. Memo 45, Stanford University, May 1968.

B & A-77 J.D. Brock and W.B. Ackerman, *An Anomaly in the Specifications of Nondeterminate Packet Systems*, MIT Laboratory for Computer Science, Computation Structures Group Note 33, 1977.

Bir-67 G. Birkhoff, Lattice Theory, American Mathematical Society Colloquium Publications, Vol. *XXV*, Amer. Math. Soc., Providence R.I., 1967.

C & O-78 R. Cartwright and D. Oppen, *Unrestricted Procedure Calls in Hoare's Logic*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson Arizona, January 1978.

Cic-76 E. Ciccarelli, *Strict Semantic Equations for Data Flow Programs*, MIT Laboratory for Computer Science, Computation Structures Group Note 26, 1976.

Den-73 J. Dennis, *First Version of a Data Flow Procedure Language*, MIT Project MAC Computation Structures Group TM-93, May 1973.

Flo-67 R.W. Floyd, *Assigning Meanings to Programs*, Proc. American Mathematical Society Symposia in Applied Mathematics, Vol. 19, 1967 (pp. 19-31).

H & P-78 D. Harel and V. Pratt, *Nondeterminism in Logics of Programs*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson Arizona, January 1978.

Hoa-69 C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM, Vol. 12, No. 12, October 1969.

Hoa-78 C.A.R. Hoare, *Communicating Sequential Processes*, Lecture at MIT, 1978.

K & M-78 G. Kahn and D. MacQueen, *Coroutines and Networks of Parallel Processes*, unpublished memorandum, 1978.

Kah-74 Gilles Kahn, *The semantics of a simple language for parallel processing*, Proc. IFIP Congress 1974, Stockholm Sweden, North-Holland, Amsterdam, 1974.

Kos-73 P.R. Kosinski, *A Data Flow Programming Language*, IBM Research Center Report RC4264, March 1973.

Kos-73b P.R. Kosinski, *A Data Flow Language for Operating Systems Programming*, ACM SIGPLAN/SIGOPS Interface Meeting, Savannah Georgia, April 1976.

Kos-76 P.R. Kosinski, *Mathematical Semantics and Data Flow Programming*, Third Annual ACM Symposium on Principles of Programming Languages, Atlanta Georgia, January 1976.

Kos-78 P.R. Kosinski, *A Straightforward Denotational Semantics for Non-determinate Data Flow Programs*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson Arizona, January 1978.

Leh-76 D.J. Lehmann, *Categories for Fixpoint-Semantics*, Ph.D. Thesis, Hebrew University of Jerusalem.

Luc-68 F.L. Luconi, *Asynchronous Computational Structures*, Ph.D. Thesis, MAC-TR49, MIT, Feb. 1968.

M & B-67 S. Maclane and G. Birkhoff, **Algebra**, The Macmillan Company, 1967.

M & R-76 G. Markowsky and B.K. Rosen, *Bases for Chain-complete Posets*, IBM Journal of Research and Development, March 1976.

M & S-76 R. Milne and C. Strachey, **A Theory of Programming Language Semantics**, Chapman and Hall, Halstead Press, 1976.

Man-74 Zohar Manna, **Mathematical Theory of Computation**, McGraw Hill, 1974.

Mar-76 G. Markowsky, *Chain-complete posets and directed sets with applications*, Algebra Universalis, 1976 (pp. 53-68).

Mar-77	G. Markowsky, *Posets, Categories, Combinatorics and Computation*, Unpublished course notes, IBM Research Center, 1977.
Plo-76	G. Plotkin, *A Powerdomain Construction*, SIAM Journal of Computing, Vol. 5 No. 3, September 1976 (pp. 452-287).
Rod-67	J.E. Rodriguez, *A Graph Model for Parallel Computations*, Ph.D. Thesis, Dept. of Electrical Engineering, MIT, 1967.
Ros-77	Barry K. Rosen, *Poset Theory of Computation*, Unpublished notes, IBM Research Center, 1977.
S & S-71	D. Scott and C. Strachey, *Toward a Mathematical Semantics for Computer Languages*, Proc. Symposium on Computers & Automata Polytechnic Institute of Brooklyn, Vol. 21, 1971 (pp. 19-46).
Sco-76	D. Scott, *Data Types as Lattices*, SIAM Journal of Computing, Vol. 5 No. 3, September 1976 (pp. 522-587).
Sim-69	H.A. Simon, **The Sciences of the Artificial**, MIT Press, Cambridge Mass., 1969.
Smy-78	M.B. Smyth, *Power Domains*, Journal of Computer and System Sciences, Vol 16, 1978 (pp. 23-26).
Sto-74	J.E. Stoy, *A Textual Language for Dataflow Programs*, Unpublished Memo, MIT Project MAC, Computation Structures Group, 1974.
Sto-77	J.E. Stoy, **Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics**, MIT Press, Cambridge Mass., 1977.

FIGURE 2.1

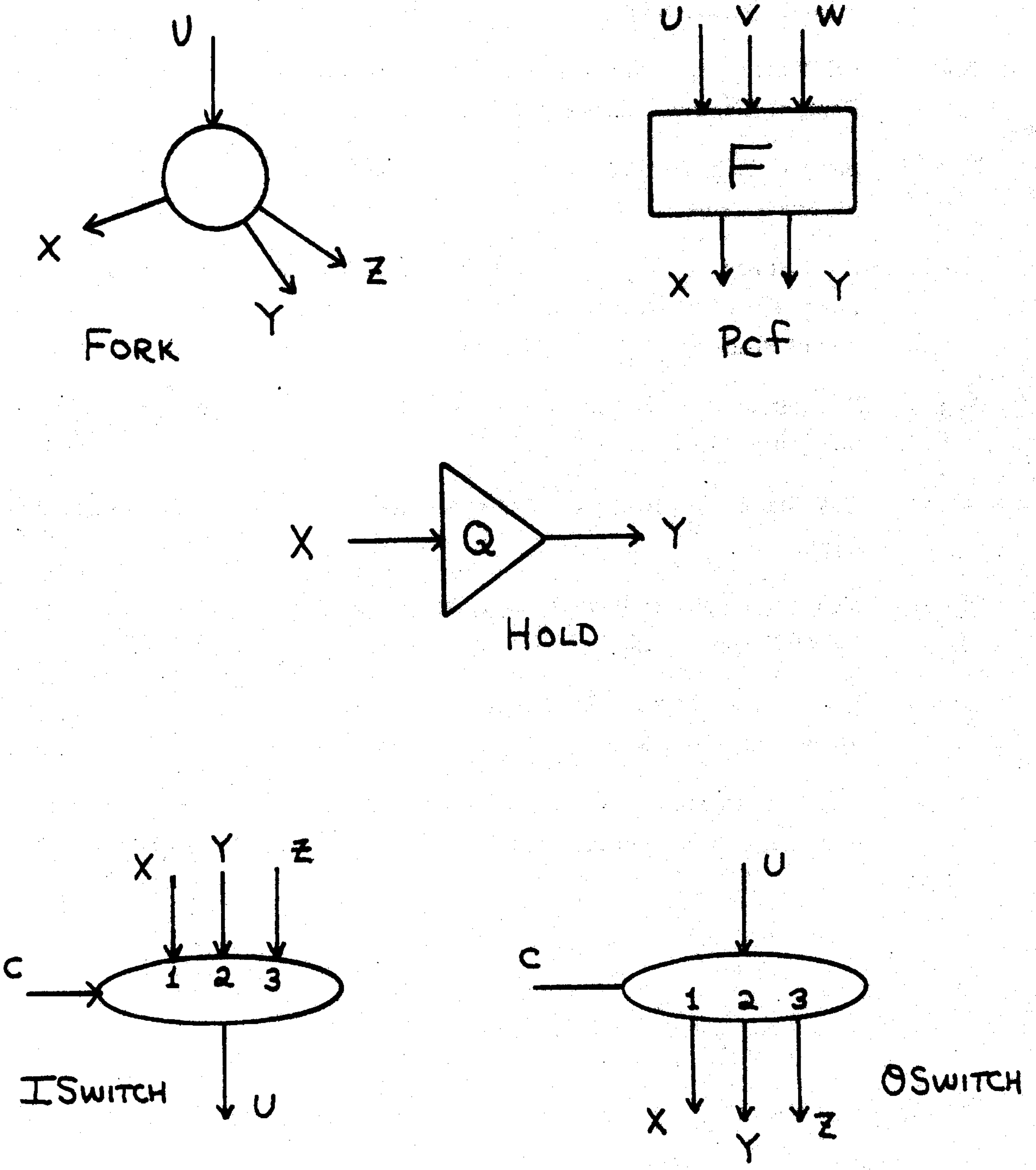

U
X
Y
Z
FORK
U
V
W
F
X
Y
Pcf
X
Q
Y
HOLD
X
Y
Z
C
1
2
3
ISWITCH
U
U
C
1
2
3
OSWITCH
X
Y
Z

FIGURE 2.2

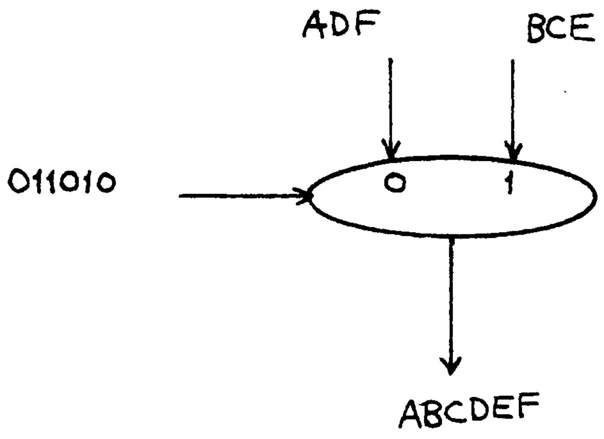


FIGURE 2.3

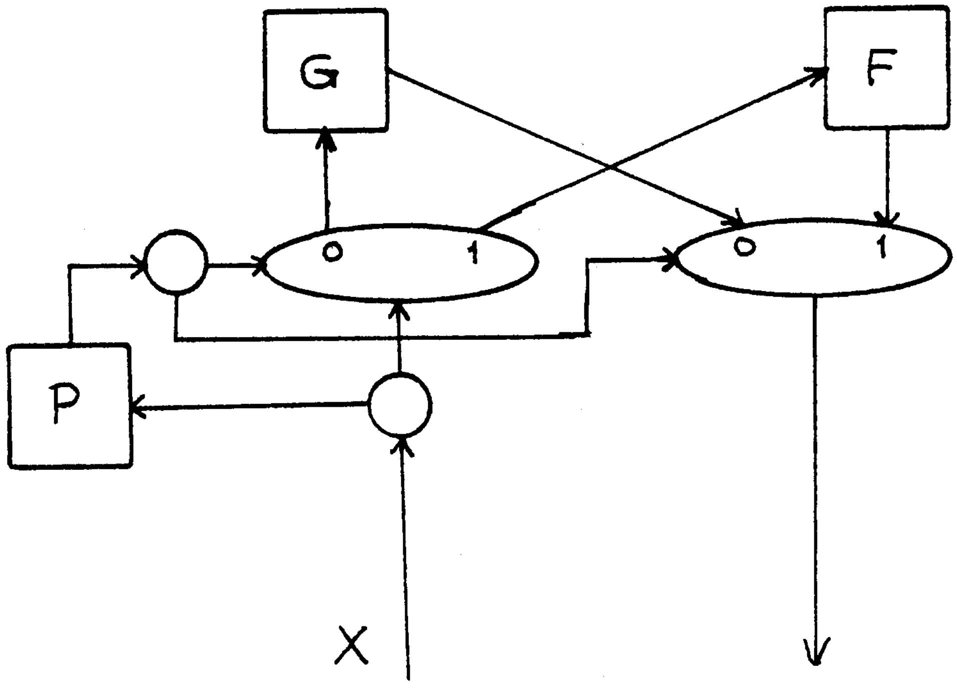


FIGURE 2.4

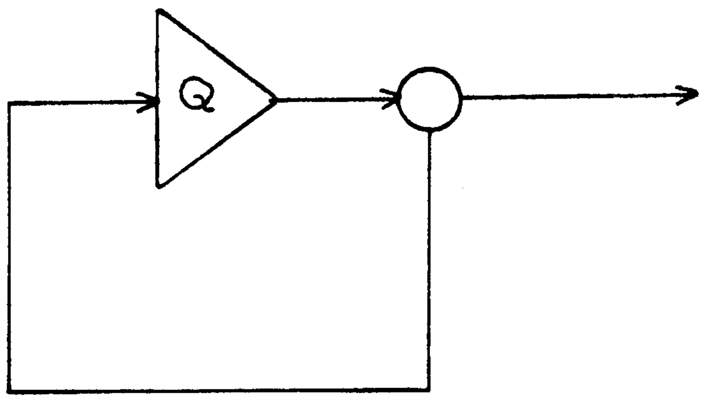


FIGURE 2.5

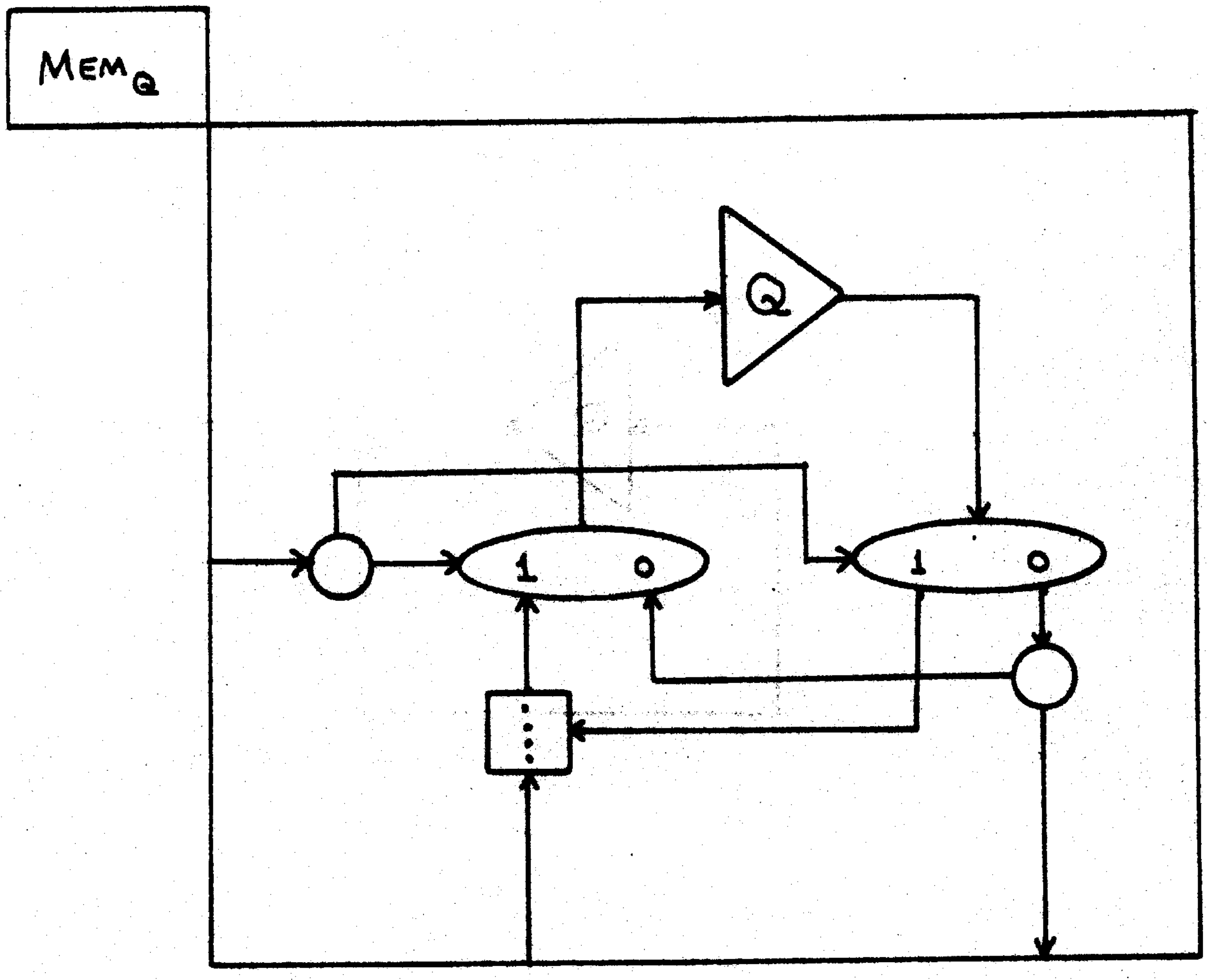


FIGURE 2.6

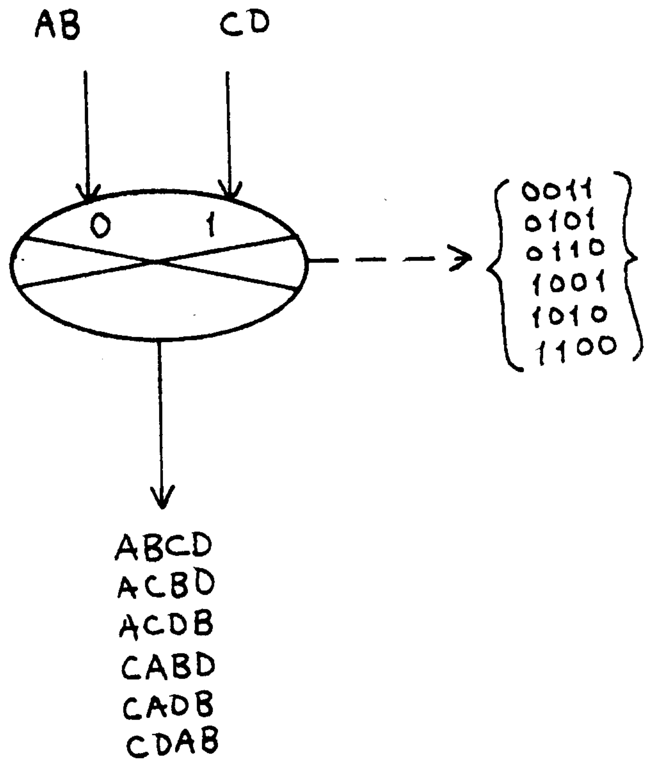

AB
CD
0
1
0011
0101
0110
1001
1010
1100
ABCD
ACBD
ACDB
CABD
CADB
CDAB

FIGURE 2.7

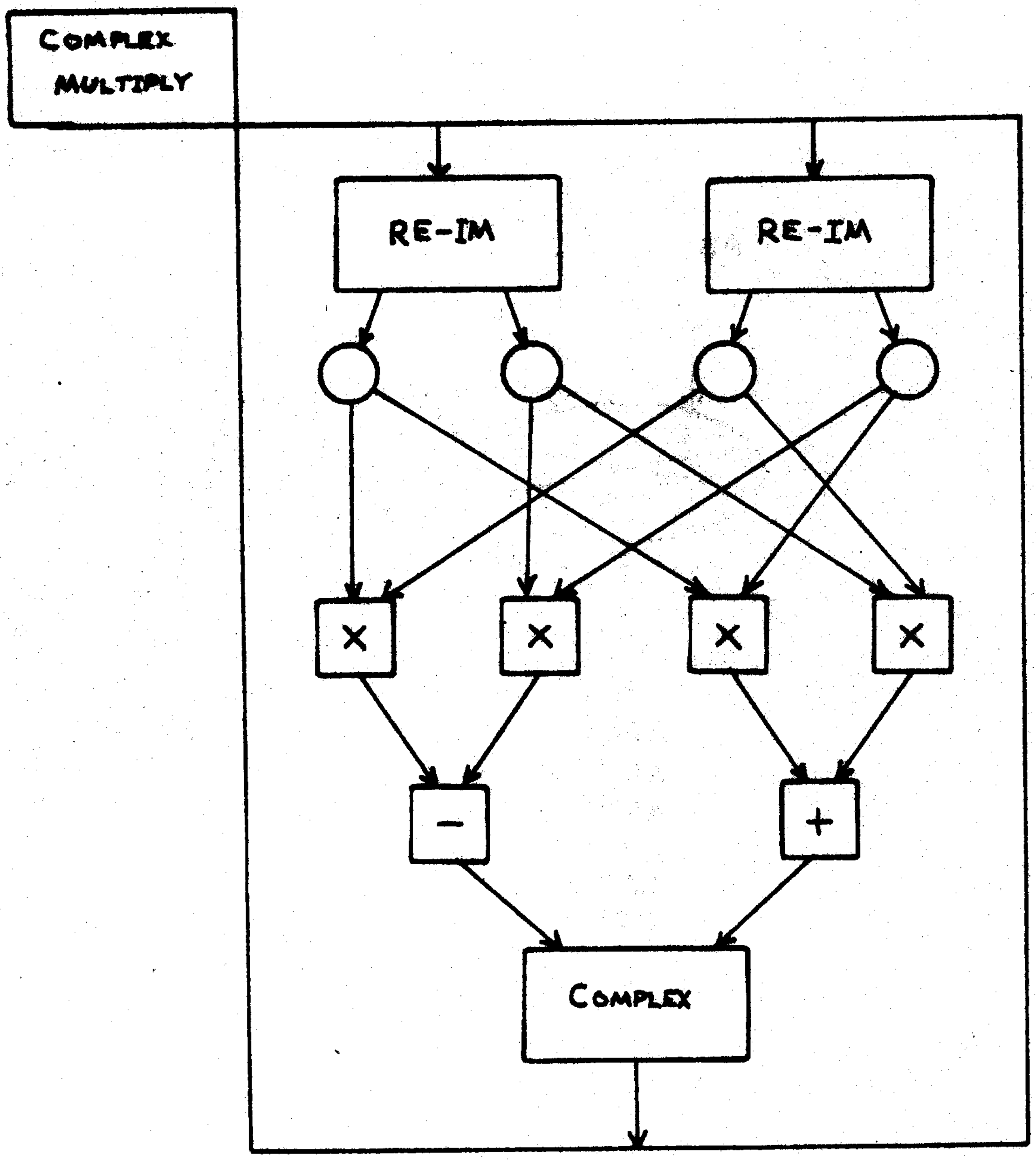


FIGURE 2.8

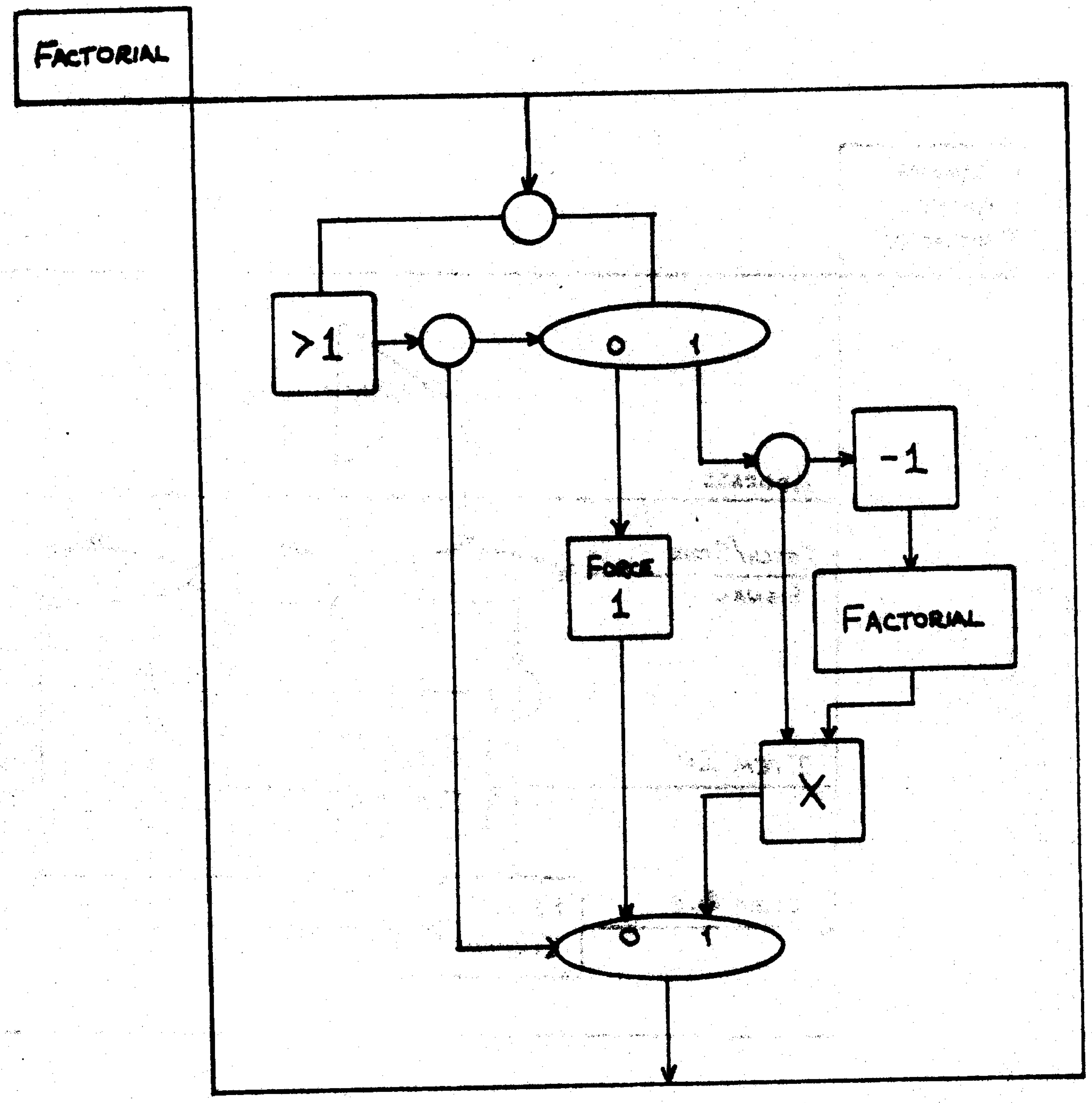


FIGURE 2.9

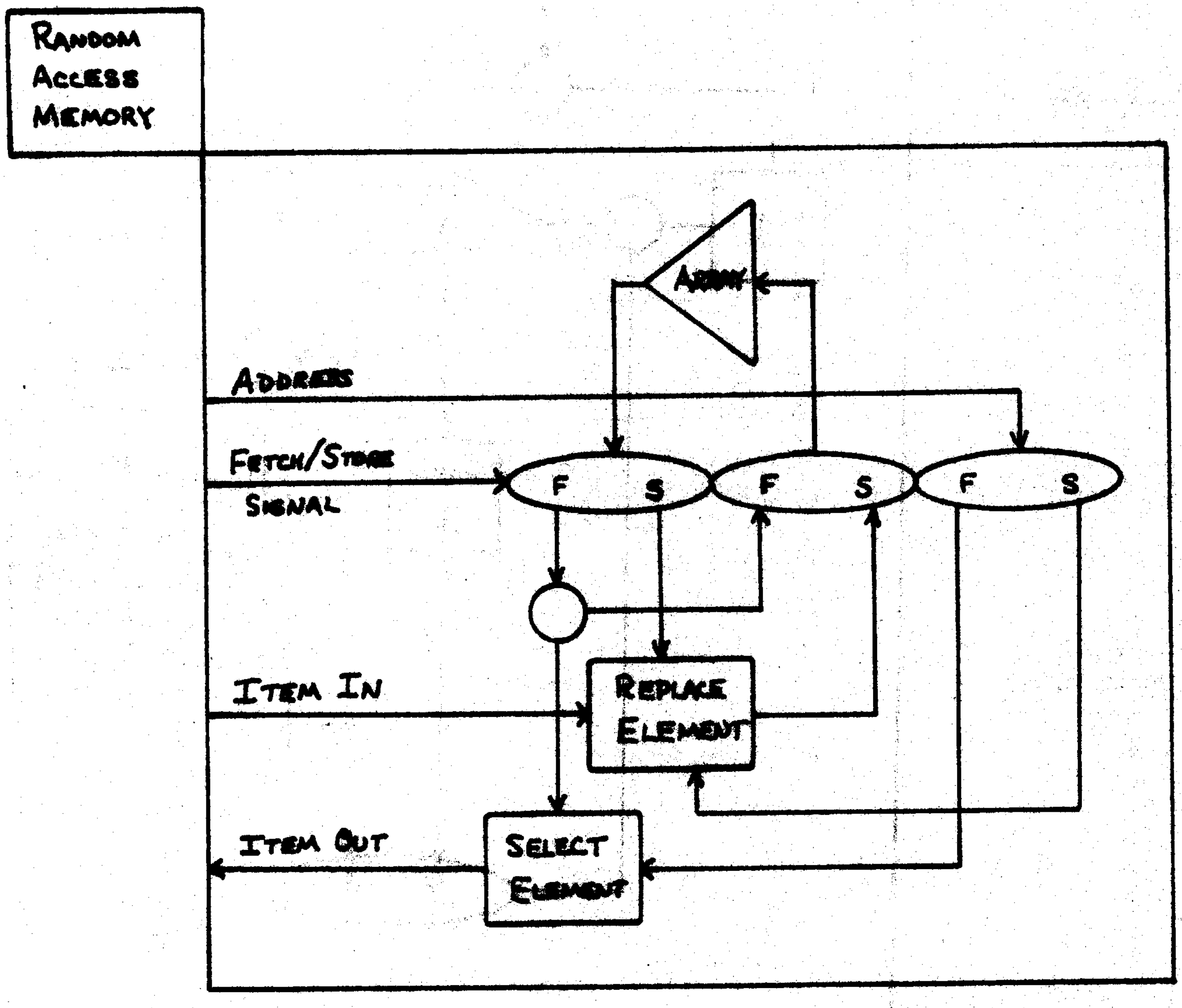

RANDOM ACCESS MEMORY
ARRAY
ADDRESS
FETCH/STORE SIGNAL
F
S
F
S
F
S
ITEM IN
REPLACE ELEMENT
ITEM OUT
SELECT ELEMENT

FIGURE 2.10

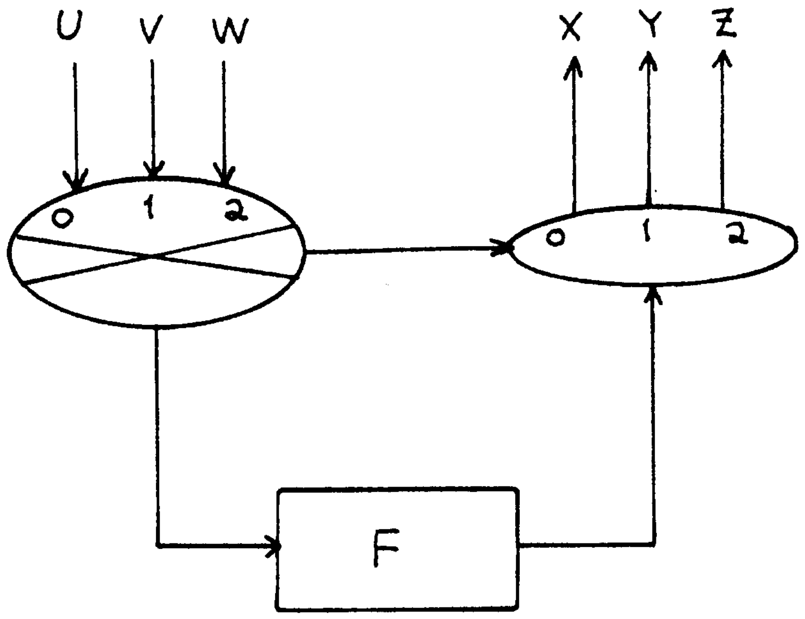


FIGURE 2.11

$$\begin{tikzcd}
U \arrow[r] & \boxed{F} \arrow[r] & X \\
V \arrow[r] & \boxed{F} \arrow[r] & Y \\
W \arrow[r] & \boxed{F} \arrow[r] & Z
\end{tikzcd}$$

FIGURE 4.1

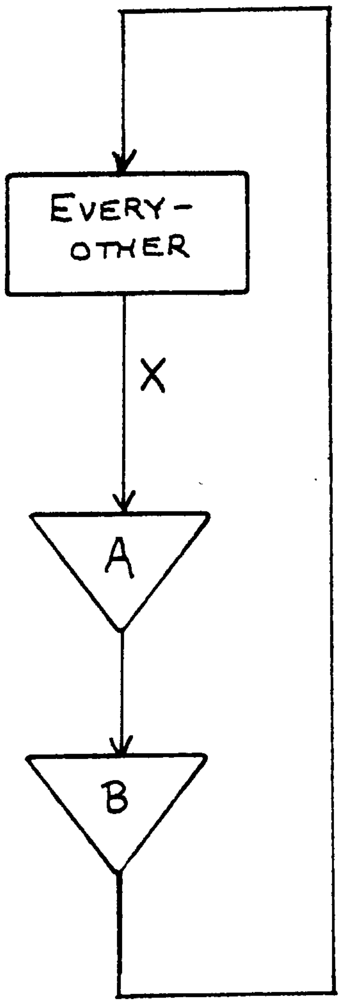


Figure 5.1

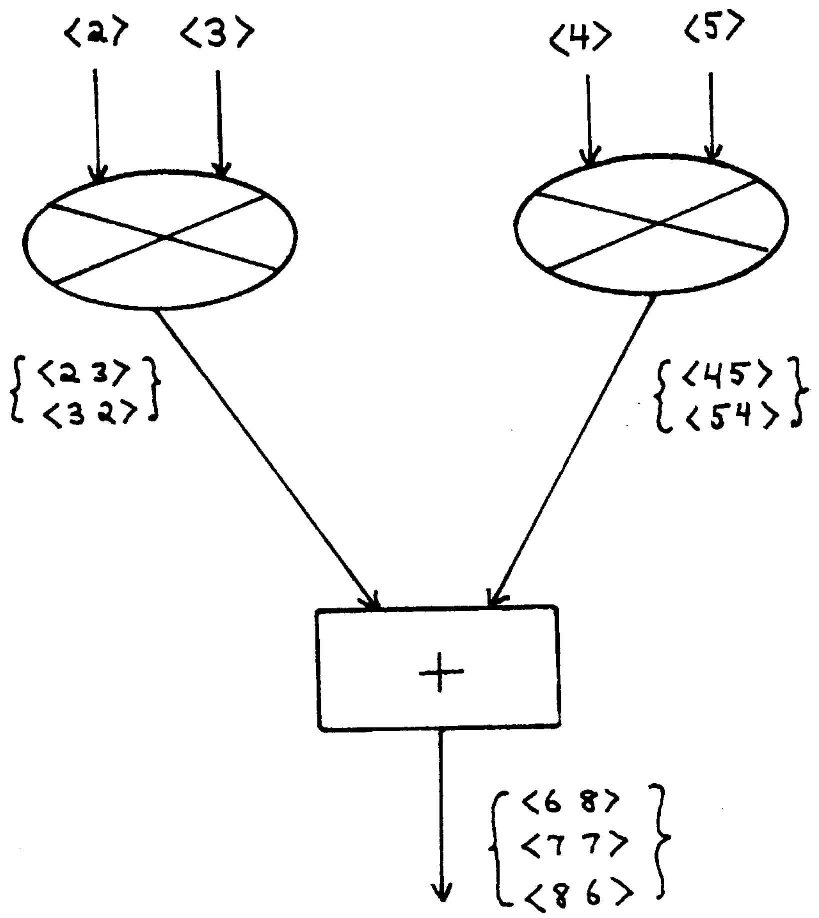


FIGURE 5.2

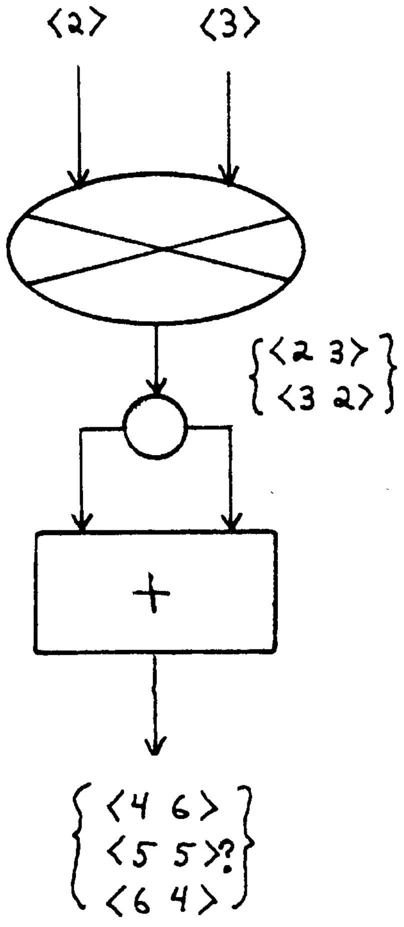


Figure 6.1

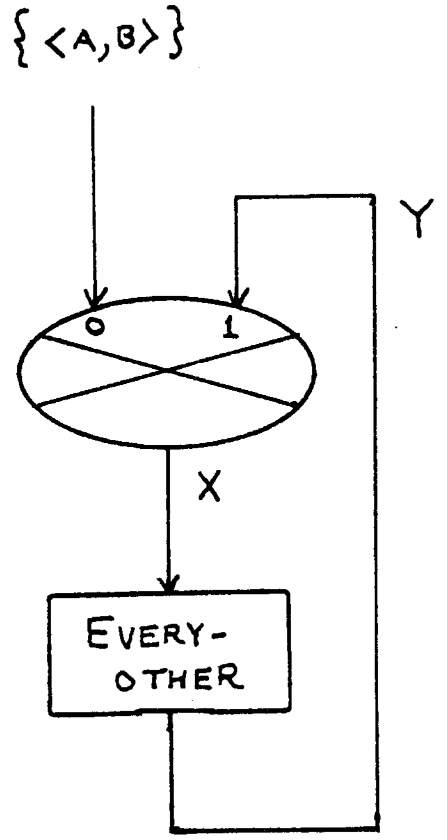


FIGURE 7.1

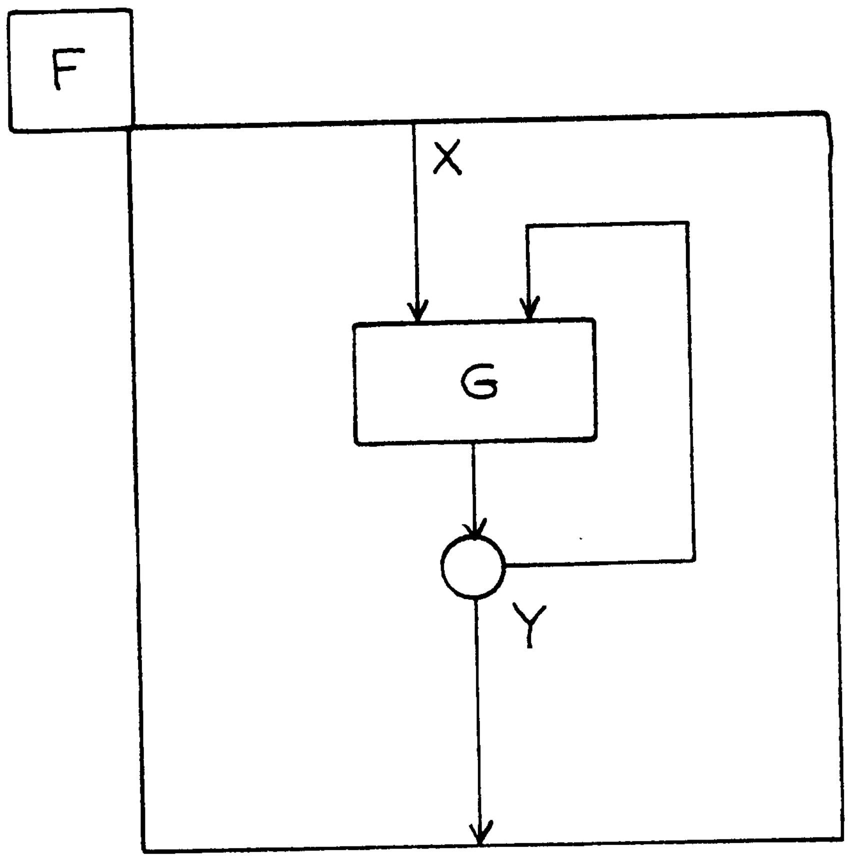
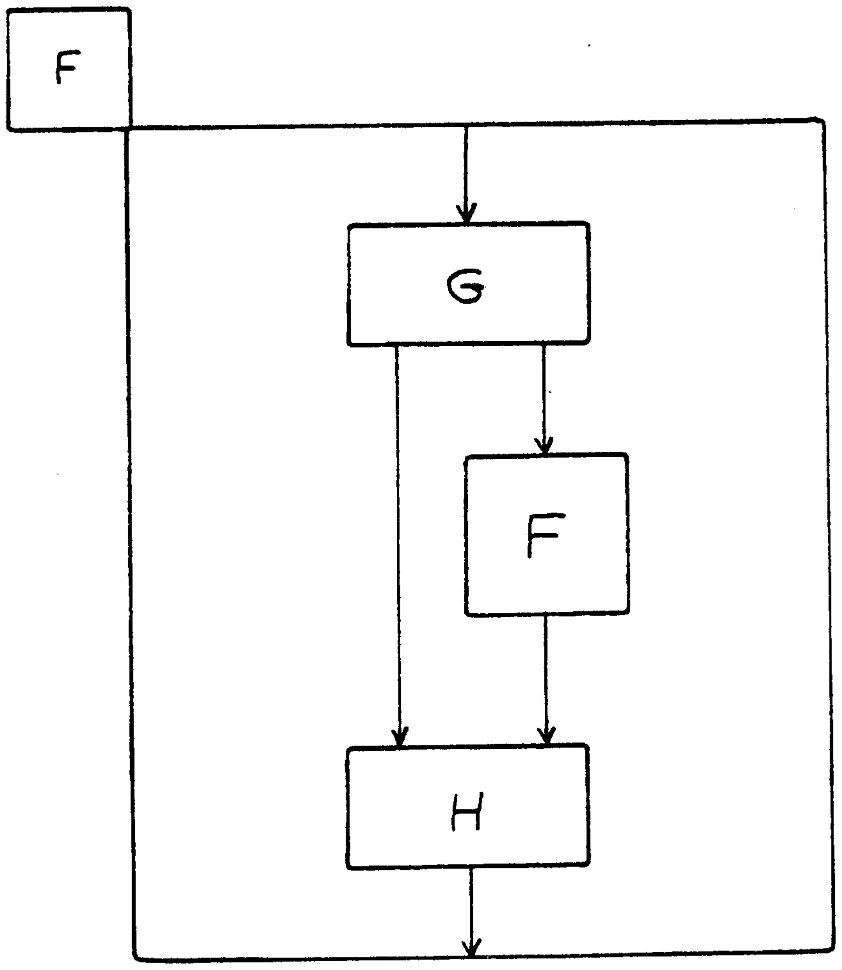


FIGURE 7.2



Biographical Note

Paul Kosinski was born in Chicago on 13 October 1942. He attended Norman Bridge public elementary school and The University of Chicago High School.

He attended the University of Chicago from 1959 to 1968, receiving the Bachelor of Science in Mathematics in 1963 and the Master of Science in Information Science in 1968. During the period 1962 to 1969 he was employed in research and advanced development at the Institute for Computer Research and Computation Center of the University. From 1964 to 1969 he also taught Computer Science at the Illinois Institute of Technology.

He spent 1969 to 1970 in the advanced operating systems group at NCR in Los Angeles. In 1970 he became a research staff member at the IBM T.J. Watson Research Center and became engaged in programming system research. In 1973 he took an educational leave to attend M.I.T. He returned to the IBM Research Center in 1976 where he continues investigating programming systems and tools.