LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-120

OPERATIONAL SEMANTICS OF A DATA FLOW LANGUAGE

Jarvis D. Brock

December 1978

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-120

OPERATIONAL SEMANTICS OF A DATA FLOW LANGUAGE

by

Jarvis Dean Brock

December 1978

This report was prepared with the support of the National Science Foundation under contract DCR75-04060; the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661, monitored by the Office of Naval Research; and a National Science Foundation graduate fellowship.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts

02139

OPERATIONAL SEMANTICS OF A DATA FLOW LANGUAGE

by

Jarvis Dean Brock

Submitted to the Department of Electrical Engineering and Computer Science
on October 2, 1978, in partial fulfillment of the requirements
for the degree of Master of Science

ABSTRACT

A data flow machine achieves high performance by the concurrent execution of machine code consisting of data flow graphs which explicitly represent the data dependencies among program instructions. This thesis presents the operational semantics of ADFL, an applicative data flow language with an iteration construct resembling tail recursion and an error-handling scheme appropriate to the concurrency of data flow. The operational semantics $\mathcal{O} \circ \mathcal{T}$ of ADFL are expressed by a two step process. The translation algorithm $\mathcal{T}$ maps an ADFL expression into its graph implementation, and the semantic function $\mathcal{O}$ maps the graph into its semantic characterization. Data flow graphs are specified by use of a graph assembly language, and the semantics of these graphs are derived by use of Kahn's fixpoint theory of communicating processes.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

Key Words: fixpoint semantics; operational semantics;
data flow programming language; applicative programming language

ACKNOWLEDGMENTS

I would like to thank Jack Dennis for introducing me to semantics and data flow and for encouraging my research in the intersection of these areas. I have learned much from my association with his research group, the Computation Structures Group of the Laboratory for Computer Science, and have enjoyed the friendship of its members. I would like to acknowledge Lynn Montz for reading a draft of this thesis and Alan Snyder for maintaining the R text justifier used to generate this document.

The Laboratory for Computer Science provided facilities for the preparation of this thesis, and the National Science Foundation provided financial support through their fellowship program.

Finally, I wish to express my gratitude to my wife Ruth for providing encouragement in my work and love and cheer in my home.

Table of Contents

1. Introduction

Recently many novel computer architectures which achieve high performance through the use of concurrency have been proposed. Most of these designs are simple variations of the von Neumann model of computation where a sequential process manipulates a memory. The effective utilization of these machines makes special demands on programmers and their programming languages, such as the structuring of data into vectors or the partitioning of programs into concurrent processes. In comparison, the data flow model of computation demands only that the principles of structured programming be followed. In this thesis we define a data flow programming language and formally specify its operational semantics.

In a data flow machine, an operation (instruction) is performed as soon as its operands have been computed. Data flow machines accept as their machine language an explicit representation of the data dependencies of program operations. Conventional computer languages designed to facilitate structured programming are easily translated into data flow machine code.

1.1 The Data Flow Model of Computation

A data flow program is represented by a directed *data flow graph* whose nodes are called *operators*. The role of operators in a data flow machine is similar to the role instructions in a von Neumann machine. The execution of an instruction corresponds to the *firing* of an operator. Each operator has several labeled input and output ports. Whenever an operator fires, it absorbs values at its input ports and produces values at its output ports. Operators have firing rules which determine when they are enabled for firing. These firing rules are based on the presence or absence

of values on the operator's ports. Most operators are enabled whenever input values are present on all input ports.

When operators are joined to form data flow graphs, the links of the graph are directed from operator output ports to operator input ports. A link transports the results produced at an operator output port to an operator input port. Thus, links form the pathways upon which data *flows* as values are absorbed and produced by the firing of operators during the execution of a graph. Unlinked operator ports within a graph are the ports of the graph itself. Graphs, like operators, absorb values at input ports and produce values at output ports.

The data flow graph for computing the distance function:

$$z = \mathrm{sqrt}((x1-x2)^2 + (y1-y2)^2)$$

is illustrated in Figure 1.1. The solid black dot in the figure represents the **copy** operator which is used to distribute the results of one output port to several input ports. Note how this graph represents operation dependencies and independencies and, consequently, the concurrency obtainable during the computation of the distance function.

1.2 Research in Data Flow Computation

There are two prerequisites to the practical use of data flow computation: (1), a machine which executes data flow graphs; and (2), a programming language which can be translated into data flow graphs. Preliminary data flow machine designs have been made by Dennis and Misunas [9] and Arvind and Gostelow [3]. Within these machines, a data flow graph is distributed over a network of processing elements. These elements operate concurrently, constrained only by the operational

dependencies of the graph. Thus, a very efficient utilization of the machine's resources appears possible.

Data flow programming languages resemble conventional languages restricted to those features whose ease of translation does not depend on the state of a computation being a single, easily manipulated entity. Because the "state" of a data flow graph is distributed for concurrency, *goto*'s, expressions with side effects, and multiple assignments to the same variable are difficult to represent. Since these "features" are generally avoided in structured programming, their absence from data flow languages is little reason for lamentation.

The "First Version of a Data Flow Language" by Dennis [7] was a rudimentary ALGOL-like language. Most data flow languages have been based on the principle of *single assignment*: Variables could be assigned only one value during a program's execution. The languages of Weng [19] and Arvind, Gostelow, and Plouffe [5], in addition to having the expressive power of ALGOL, facilitate the programming of networks of communicating processes, such as co-routines and operating systems.

The incorporation of data structure operations into data flow languages has influenced architectural designs. In theory, data flow operators using data structures would need to pass copies of entire structures among themselves. However, Ackerman [1] has specified a structure processing facility which allows pointers to structures to be passed, but still guarantees that no program observable side-effects may be caused by a structure operation. The facility is designed to process many operations concurrently.

1.3 ADFL - An Applicative Data Flow Language

ADFL, Applicative Data Flow Language, is a simplification of VAL, the value-oriented data flow language being developed by Ackerman and Dennis [2]. A BNF specification of the syntax of ADFL follows:

exp ::= *id* | *const* | *exp* , *exp* | *oper*(*exp*) | **let** *idlist* = *exp* **in** *exp* |
if *exp* **then** *exp* **else** *exp* | **for** *idlist* = *exp* **do** *iteration*

iteration ::= *exp* | **iter** *exp* | **let** *idlist* = *exp* **in** *iteration* |
if *exp* **then** *iteration* **else** *iteration*

id ::= "programming language identifiers"

idlist ::= *id* { , *id* }

const ::= "programming language constants"

oper ::= "programming language operators"

The most elementary expressions of ADFL are identifiers and constants. Tuples of expressions are also expressions. One such expression is "x, 5". The application of an operator to an expression is an expression. Although, the BNF specification only provides for operator applications in prefix form, such as "+(x, 5)"; applications in infix form, such as "x + 5", are considered acceptable equivalents (sugarings) and will be used in example ADFL programs. All operators of ADFL are required to be determinate and therefore characterizable by mathematical functions. We will not attempt to completely specify the class of operators and constants. It is assumed that at least the usual arithmetic and boolean operators and constants are present.

Since ADFL is applicative, it provides for the binding, rather than the assignment, of identifiers. Evaluation of the binding expression:

```
let y, z = x + 5, 6 in y * z
```

implies the evaluation of "y * z" with y equal to "x + 5" and z equal to 6. The result of binding is local: the values of y and z outside the binding expression are unchanged.

ADFL contains a conventional conditional expression, but has an unusual iteration expression. The evaluation of the iteration expression:

for *idlist* = *exp* do *iteration*

is accomplished by first binding the *iteration identifiers*, the elements of *idlist*, to the values of *exp*. Note from the BNF specification of *iteration*, that the evaluation of the *iteration body* will ultimately result in either an expression or the "application" of a special operator iter to an expression. This application to iter is actually a tail recursive [17] call of the iteration body with the iteration identifiers bound to the "arguments" of iter. The iteration is terminated when the evaluation of the iteration body results in an ordinary, non iter, expression. The value of this expression is returned as the value of the iteration expression. The following iteration expression computes the factorial of n:

```
for i, y = n, 1
    do if i > 1 then iter i - 1, y * i else y
```

In conventional languages execution exceptions, such as divide by zero errors, are generally handled by program interrupts. This solution is inappropriate for data flow since there is no control flow to interrupt. In ADFL execution exceptions are handled by generating special error values. For example, evaluation of "5/0" yields the special value "DivideByZero". We will not attempt to specify a large class of error values and the result of operator application to error values here. A detailed specification of this method of error-handling is given in the documentation of VAL [2]. Only one error value, err, will be specifically used in the formalism contained in this

thesis. The result of evaluating conditional expressions or iteration bodies in which the predicate is neither **true** or **false** is a tuple containing **err** as each component.

1.4 The Semantics of Data Flow Programs

The *operational* semantics of a data flow program is a formal simulation of the execution of the program's data flow graph. The formal modeling of graph execution is non-trivial. The "state" of an executing graph may be considered a *snapshot* of the tokens contained on the links of the graph [7]. The firings of operators transform the graph through an *execution sequence* of snapshots. Since most adjacent pairs of operator firings are independent, their places within an execution sequence may be interchanged. Thus, many execution sequences may represent the "same" computation.

During a graph execution, an operator receives at each input port a *history*, possibly empty, of input values and produces at each output port a history of output values. An operator is *determinate* if, for every tuple of input histories, one for each input port, a *unique* tuple of output histories is produced. The mapping from input history tuples to output history tuples is the *history function* of the determinate operator. The determinacy of an operator cannot depend on the relative timing of values received on different input ports. Operators with time-dependent behavior have non-determinate races at their input ports.

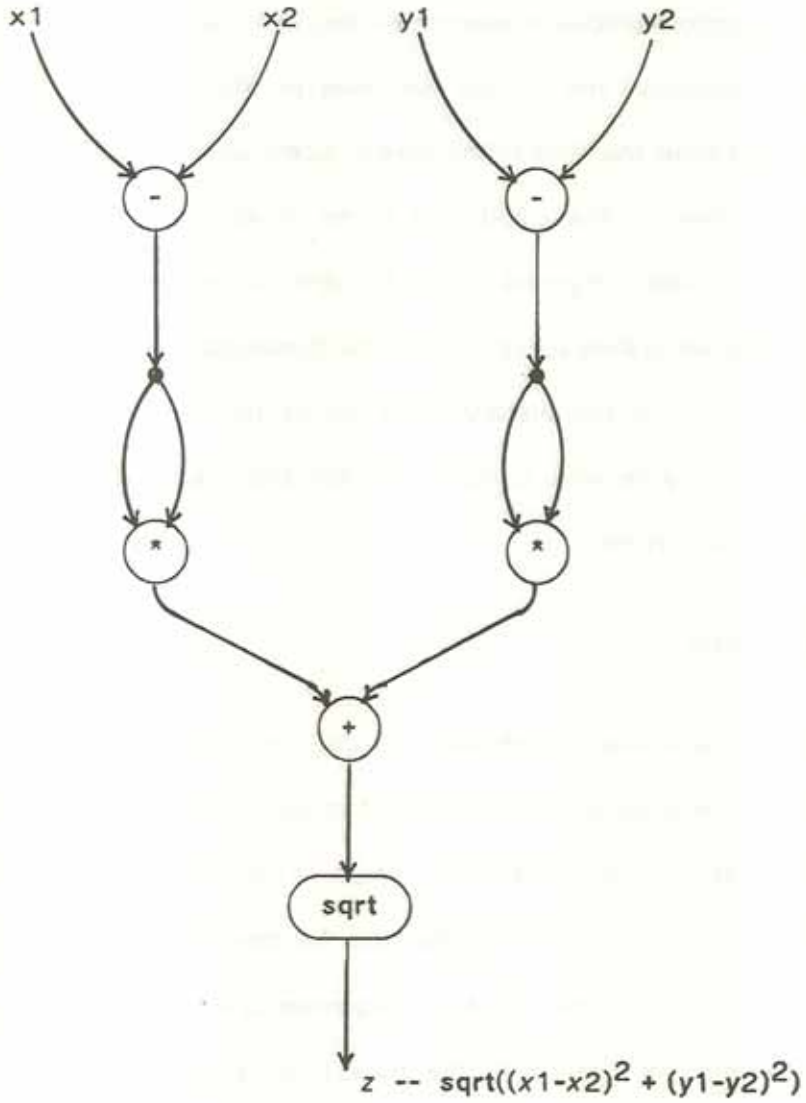
Patil [15] proved that, if all operators of a graph are determinate, the graph itself is determinate. Thus, given inputs to a determinate graph, it is necessary to examine only one execution sequence to derive the result of graph execution. Kahn [12] further simplified this derivation by noting that every graph operator,

through its history function, places a relation on the histories of its input and output links and that, consequently, the history functions of the operators form a set of simultaneous equations over the links of the graph. Using Scott's [16] *fixpoint* theory, Kahn was able to distinguish the solution of these equations which corresponds to graph execution. Because programs of ADFL are determinate, the operational semantics of ADFL may be stated by specifying the translation algorithm from programs to graphs and by specifying the history functions of the operators used in these graphs. Kahn's theory may be used to obtain the function implemented by a data flow graph following operational rules.

1.5 Synopsis of Thesis

This thesis contains a definition of the operational semantics of ADFL. These semantics may be used in formal proofs of properties of ADFL expressions and their graph implementations. In Chapter 2, a *graph assembly language* for specifying data flow graphs is introduced, and the algorithm for translating ADFL programs into data flow graphs is described. The translation algorithm is a function $\mathcal{T}$ mapping ADFL expressions into graphs. In Chapter 3, the data flow operators used to implement ADFL programs are specified, the semantic function $\mathcal{O}$ mapping graphs into their history function specification of execution is derived by use of Kahn's theory, and the operational semantics of an ADFL iteration expression is derived by application of $\mathcal{O} \circ \mathcal{T}$ to the expression. Conclusions and suggestions for future research are contained in Chapter 4.

Figure 1.1. Distance Data Flow Graph



2. $\mathcal{T}$: The Translation Algorithm

The translation algorithm is the "compiler" of ADFL. It is a function $\mathcal{T}$ which maps ADFL expressions into data flow graphs. In this chapter a language for constructing data flow graphs is described and then used to specify the translation process. Since the primary goal of this thesis is stating the operational semantics of ADFL, many ordinary compiler features such as compile-time type checking are ignored here.

Formally, a data flow operator is a history function from tuples of histories to tuples of histories. The complete specification of the semantics of the various data flow operators is given in Chapter 3. In this chapter, concerned solely with the connection of operator ports to form graphs, it suffices to be able to determine the labels of the ports of each operator. This information is available through two functions. Given a data flow operator *o*, IN(*o*) is the set of labels of the input ports of *o*, OUT(*o*) is the set of labels of the output ports. It will be assumed that the ports of data flow operators corresponding to ADFL operators are labeled by consecutive integers. For example, IN(+) = {1, 2} and OUT(+) = {1}. The functions #IN, such that #IN(*o*) is the cardinality of IN(*o*), and #OUT, defined similarly, will also be used in graph construction.

A data flow graph has four constituent parts: operators, input ports, output ports, and links. The links of a data flow graph join the input and output ports of its operators. Formally, operator interconnection may be described as a relation on operator ports. The unlinked operator ports are the ports of the graph itself. Graph ports, like operator ports, are labeled. Therefore, unlinked operator ports must be assigned the labels they assume as graph ports. Graph port assignment may be

described as a relation on graph ports and operator ports. Unfortunately, the direct description of operator interconnection and graph port assignment by relations is difficult to write and comprehend. For this reason, graphs generated by the translation process will be specified by use of a *graph assembly language.*

2.1 A Graph Assembly Language

The graph assembly language is based on the *structural description language* of Ellis [10]. It is not a complete data flow programming language. It contains only those features needed to provide a convenient and adequate specification of the translation process.

There are four components to each graph description of the graph assembly language. The first three name the input ports, output ports, and links of the graph. The fourth specifies the operators of the graph and their interconnection. Syntactically, the form of a graph description is:

inputs: ...
outputs: ...
links: ...
operators:
...

The ellipses in the above form are filled by appropriate lists. The **inputs** and **outputs** lists contain the labels of the graph input and output ports. These are the only lists available outside the graph definition. The functions **IN** and **OUT** are extended to graphs by defining **IN** to map each graph into the set of its input port labels, as given by the **inputs** list, and **OUT** to map each graph into the set of its output port labels.

The **links** and **operators** lists specify operator interconnection and graph port assignment. All links of the graphs are labeled and enumerated in the **links** lists.

Operator ports are connected by being *assigned* to the same link. These assignments are made in the **operators** list. One and only one operator output port and at least one operator input port must be assigned to a link. If more than one input port is assigned, a **copy** operator is used to distribute the outputs to all the input ports.[1] The assignment of an unlinked operator port to its graph port label is also made in the **operators** list.

The operators of the graph and the assignment of operator ports to graph links and ports are given in the **operators** list. For each instance of an operator *o* the **operators** list contains the element:

o **inputs:** ...
outputs: ...

The assignment of a graph link or input port α to an operator input port *a* is indicated by including

$$\alpha \rightarrow a$$

in the **inputs** list. The assignment of an operator output port *a* to a graph link or output port α is indicated by including

$$a \rightarrow \alpha$$

in the **outputs** list. The arrow $\rightarrow$ *always* points in the direction of data flow. This convention allows a graph input port and a graph output port to share the same label without any ambiguity of assignment occurring. It is occasionally necessary or convenient to connect graph ports and links. This is done by including an assignment as a separate element of the **operators** list. Note that by use of the functions **IN** and **OUT**, it is possible to determine if proper assignments are made for all operator ports.

---

1. The link of Dennis [7] is the **copy** operator used for this purpose.

In Figure 2.1, a graph description which generates the distance data flow graph of Chapter 1 is given. Note how the connection of the output port of the leftmost - operator through a **copy** operator to both input ports of the leftmost * operator is specified. First, the connecting link is labeled "α_1". The inclusion of the assignments "$1 \rightarrow \alpha_1$" in the **outputs** list of the - operator and "$\alpha_1 \rightarrow 1$" and "$\alpha_1 \rightarrow 2$" in the **inputs** list of the * operator completes the specification.

The translation algorithm is defined recursively. The data flow graph of an expression is constructed from the graphs of its sub-expressions. Three extensions are made to the graph assembly language to allow recursive graph definitions.

First, it must be possible to construct graphs which have smaller graphs as their components. This can be done with the present syntax for graph definitions because graphs and operators have the same external interface. Both receive values at labeled input ports and produce values at labeled output ports. Within a graph definition the ports of component subgraphs can be assigned to graph links or ports in the same manner ports of component operators are assigned.

Second, it must be possible to write graph definition schemas which incorporate subgraphs with various external characteristics. The labels of the ports of a graph or operator can be obtained with the functions **IN** and **OUT**. Assignments may be made to these ports with *range constructors*. A range constructor of the form:

$(a \in set)\ item$

specifies a list which, for every element of *set*, has an occurrence of *item* with *a* replaced by that element. For example:

$(i \in 1 \,..\, 5)\ i \rightarrow \alpha_i$

is equivalent to:

$1 \rightarrow \alpha_1, 2 \rightarrow \alpha_2, 3 \rightarrow \alpha_3, 4 \rightarrow \alpha_4, 5 \rightarrow \alpha_5$

Generally, range constructors are used to construct lists over ranges of integers or sets of input or output port labels. A similar constructor is used by Hoare [11] to specify systems of communicating processes.

And last, it must be possible to prevent the application of graph definition schemas in certain anomalous situations. For example, the ADFL expression:

if x, y **then** ... **else** ...

is invalid and little is gained by specifying its translation into a data flow graph. A graph definition schema may be restricted by the addition of a fifth top-level component of the form:

restriction: *predicate*

The schema is appropriate only in situations where the **restriction** predicate is true.

In Figure 2.2, a recursive definition of a graph $+_n$ for adding n numbers is given. The graph, an inverse binary tree of + operators, is recursively generated with two $+_{n/2}$ subgraphs and one + operator. The n graph input ports are evenly divided between the two $+_{n/2}$ subgraphs. The results of these two subgraphs are summed by the + operator. The definition is restricted to those cases where n is a power of two greater than two. Presumably, $+_2$ is the usual + operator.

2.2 The Structure of ADFL Graphs

The remainder of this chapter is devoted to a case by case specification of the translation function for ADFL. $\mathcal{T}$ maps expressions and $\mathcal{T}_I$ maps iteration bodies into corresponding data flow graphs. The special value **ERROR** denotes the result of the translation of invalid expressions or iteration bodies.

A graph corresponding to an ordinary expression or iteration body has an input port for each free variable of the expression or iteration body and, if needed, an input port trigger for enabling constants. For an expression *exp* which returns *n* values when evaluated, $\mathcal{F}[\![exp]\!]$ has *n* output ports labeled 1 through *n*. There are two possible results of evaluation for an iteration body, results to be re-iterated or results to be returned as the results of the containing iteration expression. The graph $\mathcal{F}_I[\![iteration]\!]$ of an iteration body *iteration* has a set of output ports for each possibility and an output port iter? which signals which possibility has occurred. The I ports, I1 through I*m*, are for values to be iterated, and the R ports, R1 through R*n*, are for values to be returned.

The translation functions $\mathcal{F}$ and $\mathcal{F}_I$ are defined recursively on the eleven cases of the BNF specification of the syntax of ADFL:

exp ::= *id* | *const* | *exp* , *exp* | *oper*(*exp*) | let *idlist* = *exp* in *exp* |
if *exp* then *exp* else *exp* | for *idlist* = *exp* do *iteration*

iteration ::= *exp* | iter *exp* | let *idlist* = *exp* in *iteration* |
if *exp* then *iteration* else *iteration*

2.3 The Translation of Expressions without Iteration

The first case $\mathcal{F}[\![id]\!]$, illustrated in Figure 2.3, is the most simple. $\mathcal{F}[\![id]\!]$ is the graph with the single input port *id*, one output port, and no operators. The input port *id* is directly connected to the output port.

The second case $\mathcal{F}[\![const]\!]$, illustrated in Figure 2.4, is almost as simple. $\mathcal{F}[\![const]\!]$ is the constant operator *const* with its operator ports assigned to graph ports with the same labels. The operator *const* and, consequently, the graph $\mathcal{F}[\![const]\!]$ produce the output value *const* whenever a trigger value is received.

The graph $\mathcal{T}[\![exp1, exp2]\!]$, defined in Figure 2.5, contains two subgraphs, $\mathcal{T}[\![exp1]\!]$ and $\mathcal{T}[\![exp2]\!]$. Input ports of either subgraph are assigned to graph input ports with the same label. The graph output ports are formed by concatenating the output ports of the component subgraphs.

$\mathcal{T}[\![oper(exp)]\!]$, shown in Figure 2.6, is formed by connecting the output ports of $\mathcal{T}[\![exp]\!]$ to the input ports of *oper*. If the two sets of ports do not match, $\mathcal{T}[\![oper(exp)]\!]$ is ERROR.

The free variables of the expression "**let** *idlist* = *exp*1 **in** *exp*2" are the free variables of *exp*1 plus the free variables of *exp*2 not appearing in *idlist*. The graph $\mathcal{T}[\![$**let** *idlist* = *exp*1 **in** *exp*2$]\!]$, illustrated in Figure 2.7, is constructed by connecting the *i*'th output port of $\mathcal{T}[\![exp1]\!]$ to the input port of $\mathcal{T}[\![exp2]\!]$ labeled by the *i*'th identifier of *idlist*. Input ports of $\mathcal{T}[\![exp2]\!]$ unlabeled by an identifier of *idlist* and all input ports of $\mathcal{T}[\![exp1]\!]$ are assigned to graph input ports. The output ports of $\mathcal{T}[\![exp2]\!]$ are assigned to the graph output ports. If the length of *idlist* does not match the number of output ports of $\mathcal{T}[\![exp1]\!]$ or if some identifier in *idlist* is unused in $\mathcal{T}[\![exp2]\!]$, $\mathcal{T}[\![$**let** *idlist* = *exp*1 **in** *exp*2$]\!]$ is ERROR.

The graph description of Figure 2.8 of the implementation of the conditional expression "**if** *exp*1 **then** *exp*2 **else** *exp*3" is one of the more complicated. Three subgraphs, $\mathcal{T}[\![exp1]\!]$, $\mathcal{T}[\![exp2]\!]$, and $\mathcal{T}[\![exp3]\!]$, are contained in this graph. Since the predicate *exp*1 determines which of *exp*2 and *exp*3 is selected for evaluation, the enabling of the result expression subgraphs, $\mathcal{T}[\![exp2]\!]$ and $\mathcal{T}[\![exp3]\!]$, must be controlled by the predicate subgraph, $\mathcal{T}[\![exp1]\!]$. This control is effected by connecting *gates* controlled by the predicate subgraph to the input and output ports of the result expression subgraphs.

Three gates, shown with their firing rules in Figure 2.9, are used. The T gate receives a control value from one input port, shown entering the gate horizontally, and a data value from another input port. The data value is passed to the output ports only if the control value is true. If the control value is false, the data value is simply absorbed. No output is produced. By placing a T gate controlled by the output of $\mathcal{T}[\![exp1]\!]$ on each input path of $\mathcal{T}[\![exp2]\!]$, the evaluation of *exp*2 can be restricted to when *exp*1 is true.

The role of the control value is reversed in the F gate. The data value is passed if the control value is false and is absorbed if the control value is true. F gate's control the enabling of $\mathcal{T}[\![exp3]\!]$ in the same manner T gate's control the enabling of $\mathcal{T}[\![exp2]\!]$.

The output ports of the result expression subgraphs are merged with M gate's. The M gate receives a control value which determines from which of two input ports a data value should be absorbed and produced as an output value. Each pair of output ports with the same label from the result expression subgraphs are connected to a M gate which receives the output of the predicate subgraph as its control value. The receipt of a true, respectively false, predicate value causes the data value from $\mathcal{T}[\![exp2]\!]$, respectively $\mathcal{T}[\![exp3]\!]$, to be selected. The output port of the M gate is assigned to a graph output port of the shared label.

When evaluation of the predicate yields a value other than true or false, neither of the result expressions should be evaluated and err should be generated at each graph output port. This error-handling strategy is accomplished by requiring that the T gate and the F gate absorb their data value and produce no output value and the M gate absorbs no data value and produces an err output when a control value

other than **true** or **false** is received.

$\mathcal{T}$[[**if** *exp*1 **then** *exp*2 **else** *exp*3]] is ERROR if *exp*1 returns more than one value or if *exp*2 and *exp*3 do not return the same number of values.

2.4 The Translation of Iteration Expressions

The translations of six of the seven types of expressions has been specified. Only the iteration expression remains untranslated. However, since the iteration expression contains an iteration body, it is convenient to first specify the translation of the four types of iteration bodies.

Recall that the output port **iter?** of the data flow graph of an iteration body signals whether or not output results are to be iterated or returned, the I output ports are for values to be iterated, and the R output ports are for values to be returned. The function IOUT, respectively ROUT, is defined to map a graph into the set of labels of its I, respectively R, output ports.

The graph descriptions of the iteration bodies "*exp*" and "**iter** *exp*" are given in Figures 2.10 and 2.11. The values of "*exp*" are to be returned. The values of "**iter** *exp*" are to be iterated. Consequently, in $\mathcal{T}_i$[[*exp*]] the **iter?** output value is generated with a **false** constant operator, while in $\mathcal{T}_i$[[**iter** *exp*]] it is generated with a **true** constant operator. Neither graph has a "complete" set of output ports. That is, neither contain both I and R output ports. Output port *i* of $\mathcal{T}$[[*exp*]] is assigned to output port R*i* of $\mathcal{T}_i$[[*exp*]] or to output port I*i* of $\mathcal{T}_i$[[**iter** *exp*]].

The iteration body "**let** *idlist* = *exp* **in** *iteration*" is implemented in the same manner the expression "**let** *idlist* = *exp*1 **in** *exp*2" is implemented.

The data flow graph implementation of the conditional iteration body "**if** *exp* **then** *iteration*1 **else** *iteration*2" is similar to that of the conditional expression. T gate's and F gate's, controlled by the outputs of the predicate subgraph, $\mathcal{F}[\![exp]\!]$, are placed on the input paths of the iteration body subgraphs, $\mathcal{F}_I[\![iteration1]\!]$ and $\mathcal{F}_I[\![iteration2]\!]$, so that the predicate can enable the evaluation of the selected iteration body. M gate's control the graph output ports. However, there are two complications in the use of M gate's to merge the outputs of the iteration body subgraphs. First, the selected iteration body subgraph will produce outputs at either its I or its R output ports. Consequently, the I and R output ports must be controlled separately. Second, the iteration body subgraphs do not necessarily have both I and R output ports. In Figure 2.12, $\mathcal{F}_I[\![$**if** *exp* **then** *iteration*1 **else** *iteration*2$]\!]$ is described with the assumption that both iteration body subgraphs have both I and R output ports. The modification required in other situations is given later in this section.

An IC gate is used to control the graph output ports. The IC gate has three input ports. One is connected to the output of the predicate subgraph, and the other two are connected to the iter? outputs of the iteration body subgraphs. The IC gate also has three output ports. One is assigned to the graph iter? output port, a second is the I control value for M gate's connected to the graph I ports, and a third is the R control value.

The IC gate has the following firing rule. The output of the predicate subgraph acts as a control value. It determines from which iteration body subgraph an iter? value should be absorbed. The absorbed iter? value signals whether outputs are being produced at the I or the R output ports. This value is output as the graph iter? output value and determines whether the predicate value should be transmitted as the

I or as the R control value.

If the predicate value is neither **true** or **false**, a **false** graph iter? output is generated to terminate the iteration, and **err** values are produced at the graph R ports. To accomplish this, **err** is transmitted as the R control value and **false** is transmitted through the graph **iter?** port. A formal definition of the IC gate is given in Chapter 3. The following table summarizes its firing rules.

predicate control	$\mathcal{F}_I[\![iteration1]\!]$ iter?	$\mathcal{F}_I[\![iteration2]\!]$ iter?	graph iter?	I control	R control
true	**true**	—	**true**	**true**	—
true	**false**	—	**false**	—	**true**
false	—	**true**	**true**	**false**	—
false	—	**false**	**false**	—	**false**
error	—	—	**false**	—	**err**

If both iteration body subgraphs have I output ports, these ports must match in number and must be connected through M gate's, controlled by the I control value, to the I ports of the graph. If only one subgraph has I output ports, the M gate's are omitted and the I output ports of that subgraph are assigned to the I output ports of the graph. If both subgraphs have R output ports, these ports are similarly connected to the R output ports of the graph. If only one subgraph has R output ports; E gate's, controlled by the R control value, are placed between the subgraph and graph R ports. Whenever the E gate receives a Boolean control value, it absorbs a data value and produces it as output. Whenever the E gate receives any other control value, it absorbs no data value and produces **err** as output.

All graphs described up to this point have been acyclic. If values are "dropped" in the input ports, the results will eventually "drop" out the output ports. The reader should be convinced that these graphs compute their intended functions.

These graphs can also execute in pipeline fashion. The computation of successive sets of inputs will pipeline through the graph and eventually produce successive sets of results. The computation of a later set of inputs always strictly follows a computation of an earlier set except when the computations utilize different subgraphs during the evaluation of a conditional expression or iteration body. Within the implementation of a conditional expression, the M gate's merge the output ports of such subgraphs and, using the pipelined result of predicate evaluations, restore the original order. Within the implementation of a conditional iteration body, the two computations are not necessarily merged, but the values of the iter? output port reflect the branches pursued by the computations.

The iteration expression "**for** *idlist* = *exp* **do** *iteration*" is translated, as shown in Figure 2.13, into a cyclic data flow graph containing the initialization expression subgraph, $\mathcal{T}[\![exp]\!]$, the iteration body subgraph, $\mathcal{T}_1[\![iteration]\!]$, FM gate's, and FS gate's. A FM gate is a M gate with a built-in initial control value of **false**. After "absorbing" this initial control value and passing its selected data value, the FM gate behaves like the M gate. The FS gate has one control input port and one data input port. It, too, has the built-in initial control value **false**. On receipt of a **false** control value, the FS gate absorbs a data input value, stores it in an internal register, and passes it through the gate output port. On receipt of a **true** control value, no data value is absorbed, but an output of the stored value is produced.

The sets of identifiers in *idlist*, output ports of $\mathcal{T}[\![exp]\!]$, and I output ports of $\mathcal{T}_1[\![iteration]\!]$ must all be of equal cardinality, and the identifiers of *idlist* must all be free in the iteration body. Otherwise, $\mathcal{T}[\![$**for** *idlist* = *exp* **do** *iteration*$]\!]$ is **ERROR**.

The input port of the iteration body subgraph labeled by the *i*'th identifier of *idlist* is connected to an FM gate. The input ports of this FM gate are connected to the iter? and I*i* output ports of the iteration body subgraph and the *i* output port of the initialization expression subgraph so that input is first accepted from the initialization subgraph and then is accepted from the iteration subgraph as long as the value **true** is produced at the iter? port. This corresponds to evaluating the iteration body with successive iteration body results until an ordinary, iter-less, expression is returned. The value of that ordinary expression leaves the iteration body subgraph through its R output ports. These R output ports are assigned to the graph output ports.

Every evaluation of the iteration body requires the values of its free identifiers. The values of the free identifiers not appearing in *idlist* must be generated for each iteration by FS gate's. The input ports of the iteration body subgraph labeled by these identifiers are connected to FS gate's controlled by the iter? output. Initially, the FS gate accepts, and stores, an input from a graph input port. Each time **true** is produced at the iter? port, the FS gate passes its retained value into the iteration body.

During the evaluation of an iteration expression, successive iterations need not proceed in lock-step fashion. If the iter? value is produced before all the values to be re-iterated are produced, separate iterations may pipeline through the iteration body.

When the evaluation of the iteration expression is completed, **false** is produced at the iter? port, and the FM gate's and FS gate's once again have a **false** control value with which to begin another evaluation. In Chapter 3, the data flow graph translation of the iteration expression is shown to satisfy its intended function.

All eleven cases exhausted, the specification of the translation algorithm is completed. Every expression or iteration body of ADFL has been implemented as a data flow graph with an input port for each free variable of the expression or iteration body and, optionally, an input port trigger for enabling constants. An informal description of the operational semantics of these data flow graphs has also been given. In Chapter 3, the operational semantics of data flow graphs will be formally specified. Those semantics in conjunction with the translation algorithm $\mathcal{T}$ will constitute the operational semantics of ADFL.

Figure 2.1. Distance Data Flow Graph

inputs: x1, x2, x3, x4
outputs: z
links: $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5$
operators:

- \- inputs: x1 → 1, x2 → 2
  outputs: 1 → α_1
- \- inputs: y1 → 1, y2 → 2
  outputs: 1 → α_2
- * inputs: α_1 → 1, α_1 → 2
  outputs: 1 → α_3
- * inputs: α_2 → 1, α_2 → 2
  outputs: 1 → α_4
- \+ inputs: α_3 → 1, α_4 → 2
  outputs: 1 → α_5
- sqrt inputs: α_5 → 1
  outputs: 1 → z

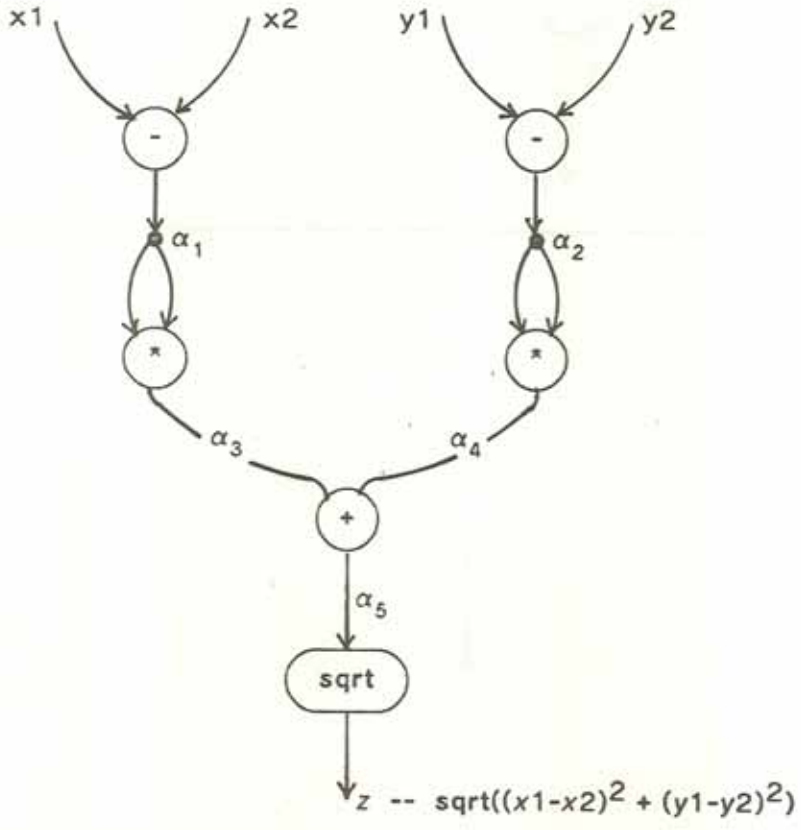


Figure 2.2. $+_n$

restriction: $n = 2^m \wedge n > 2$
inputs: $(i \in 1 \,..\, n)\ i$
outputs: 1
links: α_1, α_2
operators:
$+_{n/2}$ inputs: $(i \in 1 \,..\, n/2)\ i \rightarrow i$
outputs: $1 \rightarrow \alpha_1$
$+_{n/2}$ inputs: $(i \in 1 \,..\, n/2)\ i+n/2 \rightarrow i$
outputs: $1 \rightarrow \alpha_2$
+ inputs: $\alpha_1 \rightarrow 1, \alpha_2 \rightarrow 2$
outputs: $1 \rightarrow 1$

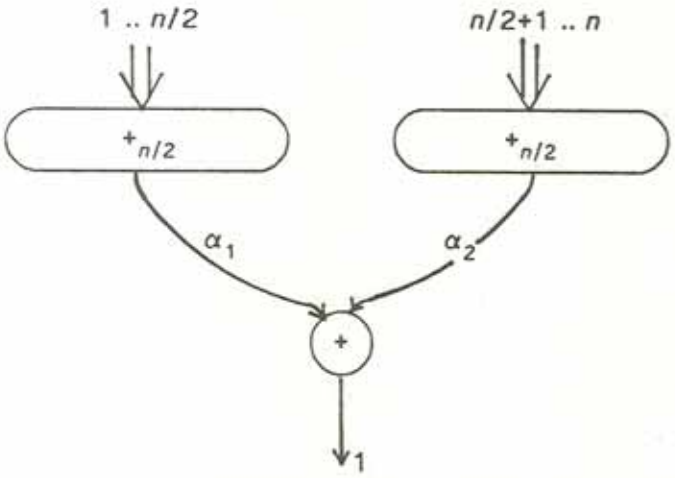


Figure 2.3. $\mathcal{I}[\![id]\!]$

inputs: *id*
outputs: 1
operators:
id → 1

Figure 2.4. $\mathcal{J}[\![const]\!]$

inputs: trigger
outputs: 1
operators:
const **inputs:** trigger → trigger
outputs: 1 → 1

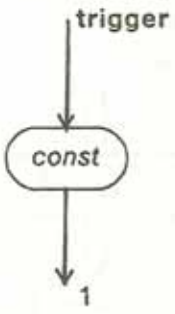


Figure 2.5. $\mathcal{J}[\![exp1, exp2]\!]$

inputs: $(a \in IN \circ \mathcal{J}[\![exp1]\!] \cup IN \circ \mathcal{J}[\![exp2]\!])\ a$
outputs: $(i \in 1\ ..\ \#OUT \circ \mathcal{J}[\![exp1]\!] + \#OUT \circ \mathcal{J}[\![exp2]\!])\ i$
operators:
$\mathcal{J}[\![exp1]\!]$ **inputs:** $(a \in IN \circ \mathcal{J}[\![exp1]\!])\ a \rightarrow a$
outputs: $(i \in OUT \circ \mathcal{J}[\![exp1]\!])\ i \rightarrow i$
$\mathcal{J}[\![exp2]\!]$ **inputs:** $(a \in IN \circ \mathcal{J}[\![exp2]\!])\ a \rightarrow a$
outputs: $(i \in OUT \circ \mathcal{J}[\![exp2]\!])\ i \rightarrow i + \#OUT \circ \mathcal{J}[\![exp1]\!]$

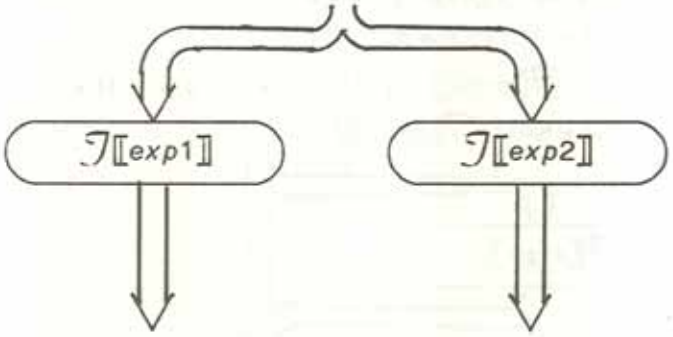


Figure 2.6. $\mathcal{F}[\![oper(exp)]\!]$

restriction: $OUT \circ \mathcal{F}[\![exp]\!] = in(oper)$
inputs: $(a \in IN \circ \mathcal{F}[\![exp]\!])\ a$
outputs: $(i \in OUT(oper))\ i$
links: $(i \in in(oper))\ \alpha_i$
operators:
$\mathcal{F}[\![exp]\!]$ inputs: $(a \in IN \circ \mathcal{F}[\![exp]\!])\ a \rightarrow a$
outputs: $(i \in OUT \circ \mathcal{F}[\![exp]\!])\ i \rightarrow \alpha_i$
oper inputs: $(i \in IN(oper))\ \alpha_i \rightarrow i$
outputs: $(i \in OUT(oper))\ i \rightarrow i$

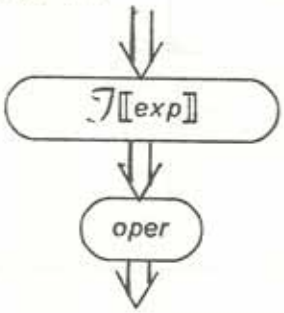


Figure 2.7. $\mathcal{F}[\![\text{let } id1, ..., idn = exp1 \text{ in } exp2]\!]$

restriction: $\#OUT \circ \mathcal{F}[\![exp1]\!] = n \wedge \{id1, ..., idn\} \subseteq IN \circ \mathcal{F}[\![exp2]\!]$
inputs: $(a \in IN \circ \mathcal{F}[\![exp1]\!] \cup (IN \circ \mathcal{F}[\![exp2]\!] - \{id1, ..., idn\}))\ a$
outputs: $(i \in OUT \circ \mathcal{F}[\![exp2]\!])\ i$
links: $(i \in 1\ ..\ n)\ \alpha_i$
operators:
$\mathcal{F}[\![exp1]\!]$ inputs: $(a \in IN \circ \mathcal{F}[\![exp1]\!])\ a \rightarrow a$
outputs: $(i \in 1\ ..\ n)\ i \rightarrow \alpha_i$
$\mathcal{F}[\![exp2]\!]$ inputs: $(a \in \mathcal{F}[\![exp2]\!] - \{id1, ..., idn\})\ a \rightarrow a, (i \in 1\ ..\ n)\ \alpha_i \rightarrow idi$
outputs: $(i \in OUT \circ \mathcal{F}[\![exp2]\!])\ i \rightarrow i$

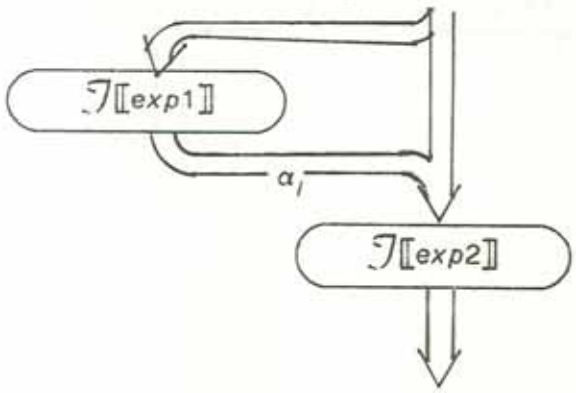


Figure 2.8. $\mathcal{F}[\![\text{if } exp1 \text{ then } exp2 \text{ else } exp3]\!]$

restriction: $OUT\circ\mathcal{F}[\![exp1]\!] = \{1\} \wedge OUT\circ\mathcal{F}[\![exp2]\!] = OUT\circ\mathcal{F}[\![exp3]\!]$
inputs: $(a \in IN\circ\mathcal{F}[\![exp1]\!] \cup IN\circ\mathcal{F}[\![exp2]\!] \cup IN\circ\mathcal{F}[\![exp3]\!])\ a$
outputs: $(i \in OUT\circ\mathcal{F}[\![exp2]\!])\ i$
links: α, $(a \in IN\circ\mathcal{F}[\![exp2]\!])\ \beta^T_a$, $(a \in IN\circ\mathcal{F}[\![exp3]\!])\ \beta^F_a$,
$(i \in OUT\circ\mathcal{F}[\![exp2]\!])\ \gamma^T_i$, $(i \in OUT\circ\mathcal{F}[\![exp3]\!])\ \gamma^F_i$
operators:

$\mathcal{F}[\![exp1]\!]$ **inputs:** $(a \in IN\circ\mathcal{F}[\![exp1]\!])\ a \to a$
outputs: $1 \to \alpha$

$(a \in IN\circ\mathcal{F}[\![exp2]\!])$ **T gate inputs:** $\alpha \to 1,\ a \to 2$
outputs: $1 \to \beta^T_a$

$(a \in IN\circ\mathcal{F}[\![exp3]\!])$ **F gate inputs:** $\alpha \to 1,\ a \to 2$
outputs: $1 \to \beta^F_a$

$\mathcal{F}[\![exp2]\!]$ **inputs:** $(a \in IN\circ\mathcal{F}[\![exp2]\!])\ \beta^T_a \to a$
outputs: $(i \in OUT\circ\mathcal{F}[\![exp2]\!])\ i \to \gamma^T_i$

$\mathcal{F}[\![exp3]\!]$ **inputs:** $(a \in IN\circ\mathcal{F}[\![exp3]\!])\ \beta^F_a \to a$
outputs: $(i \in OUT\circ\mathcal{F}[\![exp3]\!])\ i \to \gamma^F_i$

$(i \in OUT\circ\mathcal{F}[\![exp2]\!])$ **M gate inputs:** $\alpha \to 1,\ \gamma^T_i \to 2,\ \gamma^F_i \to 3$
outputs: $1 \to i$

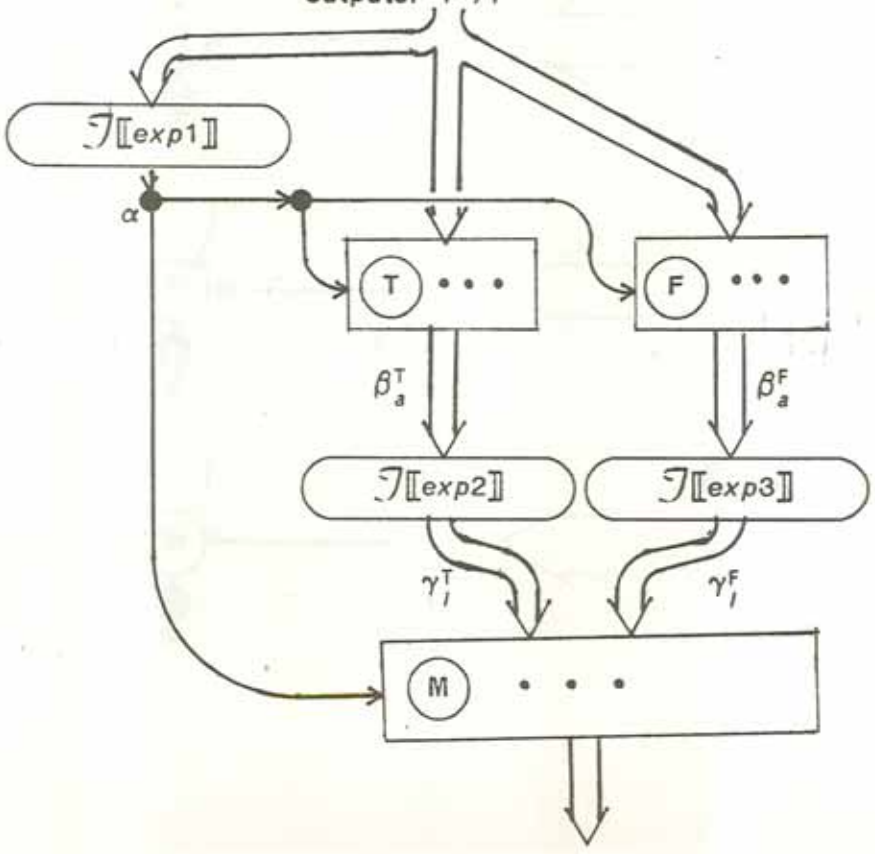


Figure 2.9. Gates for Implementing Conditional Expressions

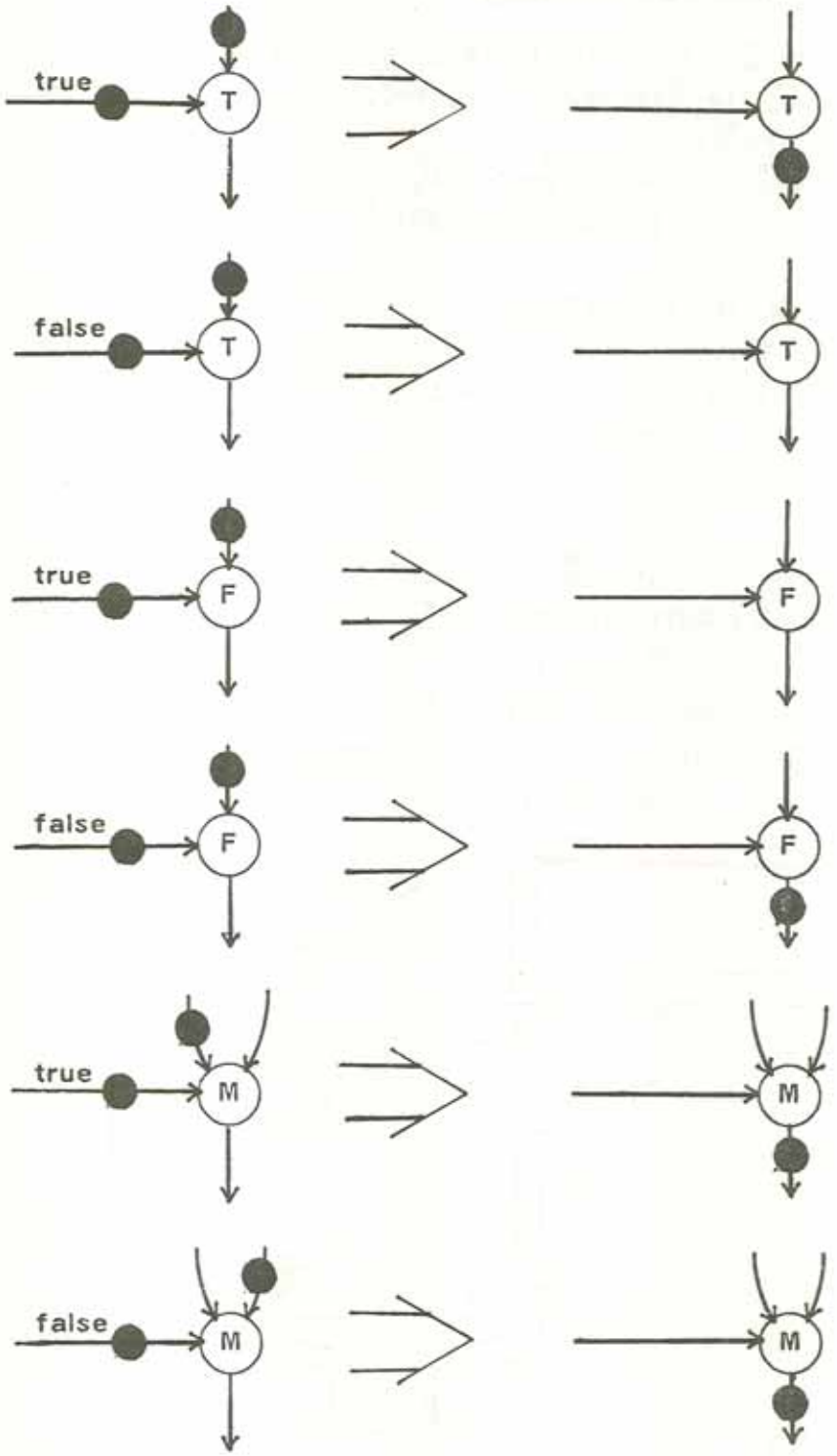


Figure 2.10. $\mathcal{F}_I[\![exp]\!]$

inputs: ($a \in IN \circ \mathcal{F}[\![exp]\!] \cup \{trigger\}$) a
outputs: ($i \in OUT \circ \mathcal{F}[\![exp]\!]$) R$i$, iter?
operators:
false inputs: trigger → trigger
outputs: 1 → iter?
$\mathcal{F}[\![exp]\!]$ inputs: ($a \in IN \circ \mathcal{F}[\![exp]\!]$) $a \rightarrow a$
outputs: ($i \in OUT \circ \mathcal{F}[\![exp]\!]$) $i \rightarrow$ Ri

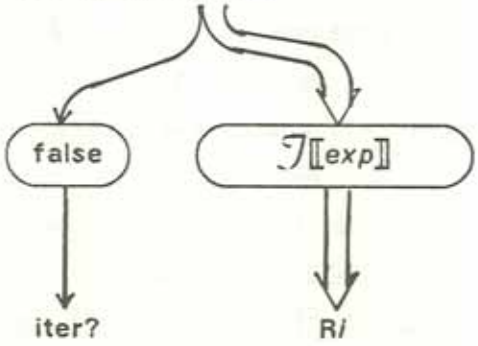


Figure 2.11. $\mathcal{F}_I[\![\text{iter } exp]\!]$

inputs: ($a \in IN \circ \mathcal{F}[\![exp]\!] \cup \{trigger\}$) a
outputs: ($i \in OUT \circ \mathcal{F}[\![exp]\!]$) l$i$, iter?
operators:
true inputs: trigger → trigger
outputs: 1 → iter?
$\mathcal{F}[\![exp]\!]$ inputs: ($a \in IN \circ \mathcal{F}[\![exp]\!]$) $a \rightarrow a$
outputs: ($i \in OUT \circ \mathcal{F}[\![exp]\!]$) $i \rightarrow$ li

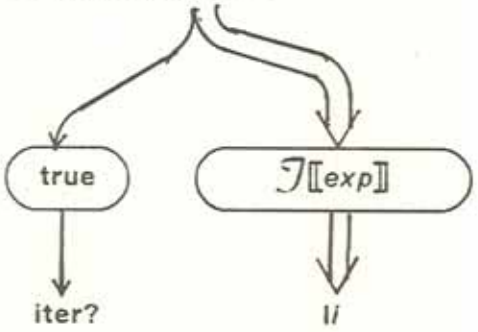


Figure 2.12. $\mathcal{F}_I[\![$if *exp* then *iteration*1 else *iteration*2$]\!]$

restriction: $OUT\circ\mathcal{F}[\![exp]\!] = \{1\} \wedge OUT\circ\mathcal{F}_I[\![iteration1]\!] = OUT\circ\mathcal{F}_I[\![iteration2]\!]$

inputs: $(a \in IN\circ\mathcal{F}[\![exp]\!] \cup IN\circ\mathcal{F}_I[\![iteration1]\!] \cup IN\circ\mathcal{F}_I[\![iteration2]\!])\ a$

outputs: $(a \in OUT\circ\mathcal{F}_I[\![iteration1]\!])\ a$

links: α, α^I, α^R, $(a \in IN\circ\mathcal{F}_I[\![iteration1]\!])\ \beta^T_a$, $(a \in IN\circ\mathcal{F}_I[\![iteration2]\!])\ \beta^F_a$,
$(a \in OUT\circ\mathcal{F}_I[\![iteration1]\!])\ \gamma^T_a$, $(a \in OUT\circ\mathcal{F}_I[\![iteration2]\!])\ \gamma^F_a$

operators:

$\mathcal{F}[\![exp]\!]$ inputs: $(a \in IN\circ\mathcal{F}[\![exp]\!])\ a$
outputs: $1 \rightarrow \alpha$

$(a \in IN\circ\mathcal{F}_I[\![iteration1]\!])$ T gate inputs: $\alpha \rightarrow 1$, $a \rightarrow 2$
outputs: $1 \rightarrow \beta^T_a$

$(a \in IN\circ\mathcal{F}_I[\![iteration2]\!])$ F gate inputs: $\alpha \rightarrow 1$, $a \rightarrow 2$
outputs: $1 \rightarrow \beta^F_a$

$\mathcal{F}[\![iteration1]\!]$ inputs: $(a \in IN\circ\mathcal{F}_I[\![iteration1]\!])\ \beta^T_a \rightarrow a$
outputs: $(a \in OUT\circ\mathcal{F}_I[\![iteration1]\!])\ a \rightarrow \gamma^T_a$

$\mathcal{F}[\![iteration2]\!]$ inputs: $(a \in IN\circ\mathcal{F}_I[\![iteration2]\!])\ \beta^F_a \rightarrow a$
outputs: $(a \in OUT\circ\mathcal{F}_I[\![iteration2]\!])\ a \rightarrow \gamma^F_a$

IC gate inputs: $\alpha \rightarrow 1$, $\gamma^T_{Iter?} \rightarrow 2$, $\gamma^F_{Iter?} \rightarrow 3$
outputs: $1 \rightarrow$ iter?, $2 \rightarrow \alpha^I$, $3 \rightarrow \alpha^R$

$(Ii \in IOUT\circ\mathcal{F}_I[\![iteration1]\!])$ M gate inputs: $\alpha^I \rightarrow 1$, $\gamma^T_{Ii} \rightarrow 2$, $\gamma^F_{Ii} \rightarrow 3$
outputs: $1 \rightarrow Ii$

$(Ri \in ROUT\circ\mathcal{F}_I[\![iteration1]\!])$ M gate inputs: $\alpha^R \rightarrow 1$, $\gamma^T_{Ri} \rightarrow 2$, $\gamma^F_{Ri} \rightarrow 3$
outputs: $1 \rightarrow Ri$

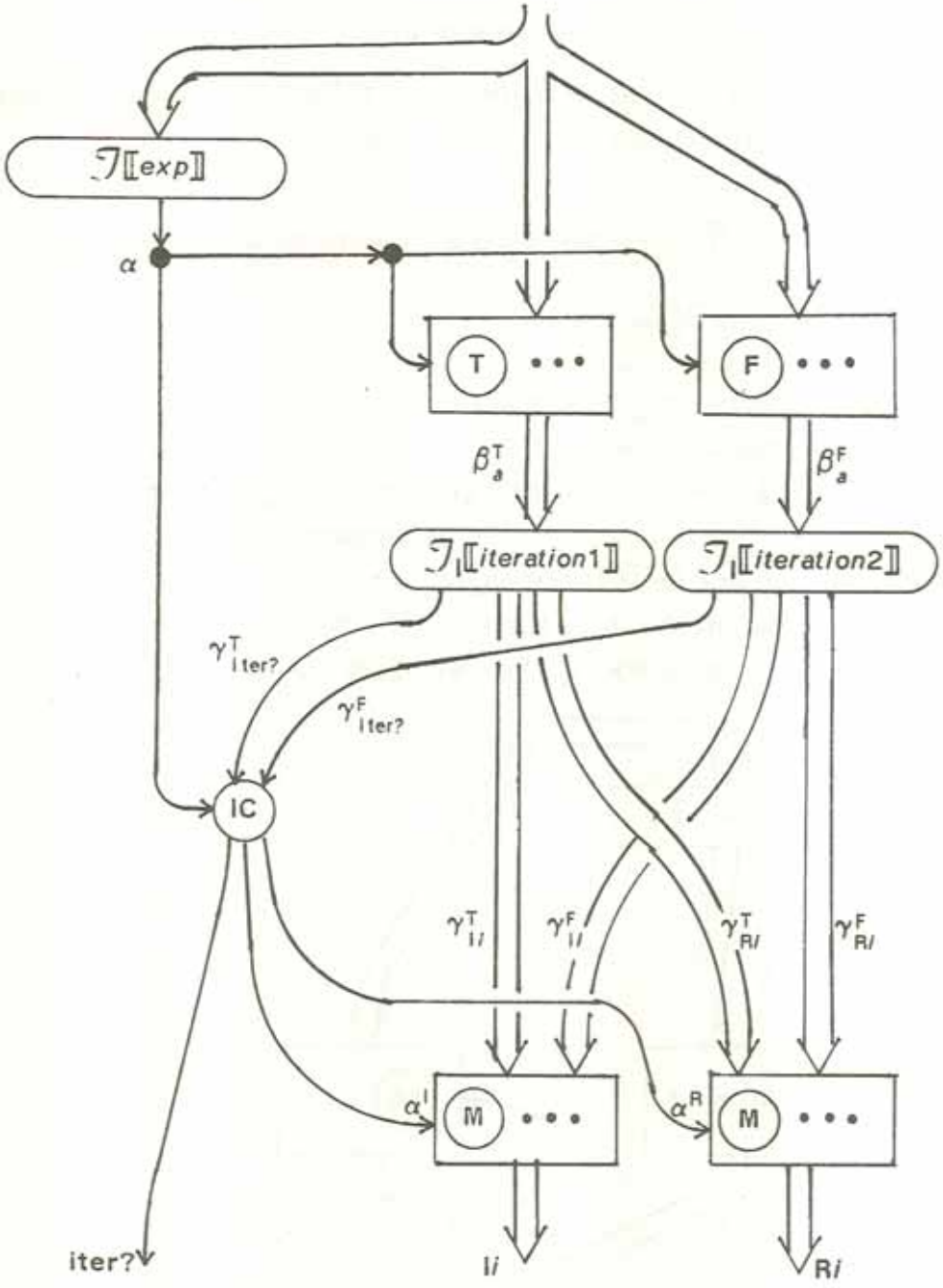

$\mathcal{T}$[[exp]]
α
T
F
β^T_a
β^F_a
$\mathcal{T}_I$[[iteration1]]
$\mathcal{T}_I$[[iteration2]]
$\gamma^T_{iter?}$
$\gamma^F_{iter?}$
IC
γ^T_{Ii}
γ^F_{Ii}
γ^T_{Ri}
γ^F_{Ri}
α^I
α^R
M
M
iter?
Ii
Ri

Figure 2.13. $\mathcal{T}[\![\text{for } id1, ..., idn = exp \text{ do } iteration]\!]$

restriction: $n = \#\text{OUT}\circ\mathcal{T}[\![exp]\!] = \#\text{IOUT}\circ\mathcal{T}_1[\![iteration]\!] \wedge \{id1, ..., idn\} \subseteq \text{IN}\circ\mathcal{T}[\![exp]\!]$

inputs: $(a \in \text{IN}\circ\mathcal{T}[\![exp]\!] \cup (\text{IN}\circ\mathcal{T}_1[\![iteration]\!] - \{id1, ..., idn\})\ a$

outputs: $(\text{R}i \in \text{ROUT}\circ\mathcal{T}_1[\![iteration]\!])\ i$

links: $(i \in 1 .. n)\ \alpha_i$, $(a \in \text{IN}\circ\mathcal{T}_1[\![iteration]\!])\ \beta_a$, $(i \in 1 .. n)\ \gamma_i$, $\gamma_{\text{iter?}}$

operators:

$\mathcal{T}[\![exp]\!]$ inputs: $(a \in \text{IN}\circ\mathcal{T}[\![exp]\!])\ a \rightarrow a$

outputs: $(i \in 1 .. n)\ i \rightarrow \alpha_i$

$(i \in 1 .. n)$ FM gate inputs: $\gamma_{\text{iter?}} \rightarrow 1, \gamma_i \rightarrow 2, \alpha_i \rightarrow 3$

outputs: $1 \rightarrow \beta_{idi}$

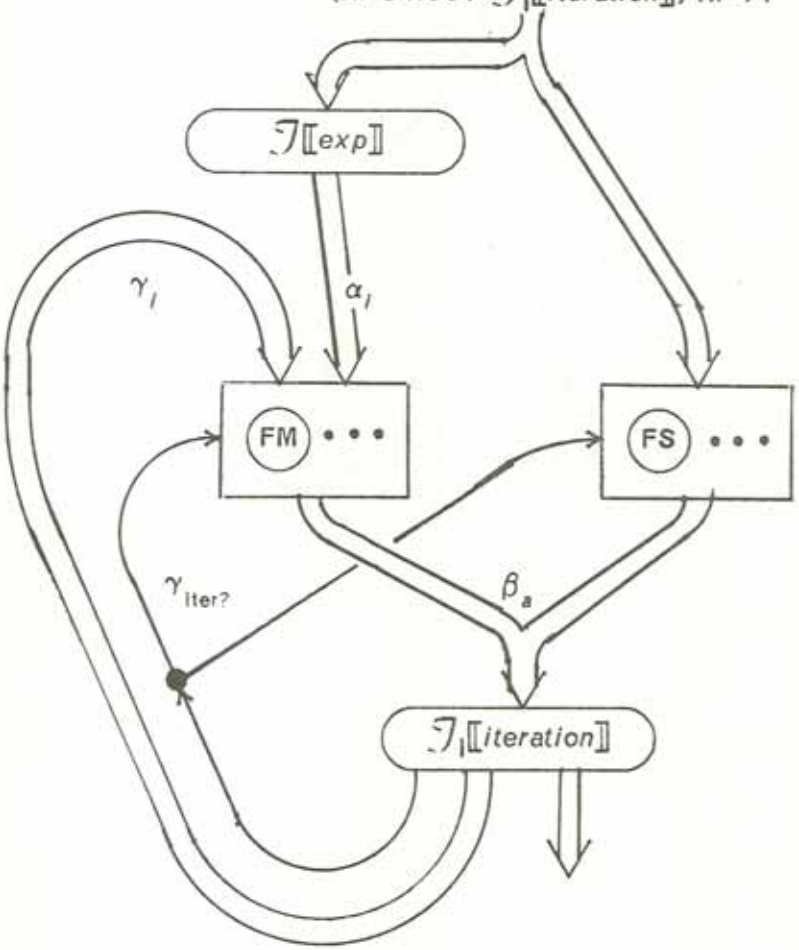
$(a \in (\text{IN}\circ\mathcal{T}_1[\![iteration]\!] - \{id1, ..., idn\}))$ FS gate inputs: $\gamma_{\text{iter?}} \rightarrow 1, a \rightarrow 2$

outputs: $1 \rightarrow \beta_a$

$\mathcal{T}_1[\![iteration]\!]$ inputs: $(a \in \text{IN}\circ\mathcal{T}_1[\![iteration]\!])\ \beta_a \rightarrow a$

outputs: iter? $\rightarrow \gamma_{\text{iter?}}$, $(i \in 1 .. n)\ \text{I}i \rightarrow \gamma_i$,

$(\text{R}i \in \text{ROUT}\circ\mathcal{T}_1[\![iteration]\!])\ \text{R}i \rightarrow i$



3. $\mathcal{O}$: The Operational Semantics

In this chapter, a function $\mathcal{O}$ mapping data flow graphs into their operational characterizations is given. Operational semantics are defined using Kahn's [12] formal model of parallel computation. Each data flow operator or graph is characterized by a history function mapping tuples of input histories into tuples of output histories. The history function of a graph is derived using the history functions of its operators. Kahn originally used his theory to characterize processes written in an ALGOL-like language augmented with get and put statements for receiving and transmitting values on queues and to derive the result of interconnecting such processes. The contrast between Kahn's level of application and ours illustrates the relative concurrency obtained with data flow and sequential control flow program execution. In Kahn's sequential control flow application, concurrency is limited to the simultaneous execution of processes consisting of several programming language statements; where, in our data flow application, concurrency occurs at even the most elementary level of expression evaluation.

The first section of this chapter describes Kahn's theory as applied to data flow graphs. The formal characterization of operators and graphs, the restrictions placed on the behavior of operators, the method for deriving the semantics of graphs, and the closure properties of this characterization of operators and graphs are given. Readers familiar with Kahn's theory may wish to proceed to the second section. The second section specifies the semantics of operators used in graph implementations of ADFL expressions. The third and final section illustrates the application of this theory to derive the semantics of an ADFL iteration expression.

3.1 Kahn's Semantics of Data Flow Graphs

The operational semantics of a data flow operator o is given by a history function $()[\![o]\!]$ mapping input history tuples into output history tuples. For each input history tuple X, representing the history of values received at the input ports of o, the output history tuple $()[\![o]\!](X)$ represents the history of values produced at the output ports of o in response to X. Input history tuple X has as its components a history, a possibly infinite sequence of values, for each port of o. Formally, input history tuple X is a function which maps each input port label a of o into the input history $X(a)$, often denoted X_a, received at that port. Output history functions are defined similarly.

Not all operators may be characterized by Kahn's history functions. In particular, only determinate operators which for each input history have only one possible output history may be characterized thusly. Since only determinate operators were used in Chapter 2 to construct graph implementations of ADFL expressions, the history function characterization is adequate for describing the operational semantics of ADFL. There are two other requirements which operators must satisfy in order that fixpoint methods may be used to determine the result of their interconnection. However, these requirements are not restrictions but rather a formal statement of some properties of which history functions of physically realizable data flow operators must satisfy.

First, the domain or range of a history function must be a complete partially ordered set with a least element. We review the definition of a complete partial order.

Definition: A relation $\sqsubseteq$ on a set A is a *partial order* if $\sqsubseteq$ is:

(1), reflexive, $\forall\, x \in A,\ x \sqsubseteq x$,

(2), antisymmetric, $\forall\, x, y \in A,\ x \sqsubseteq y \wedge y \sqsubseteq x \rightarrow x = y$, and

(3), transitive, $\forall\, x, y, z \in A,\ x \sqsubseteq y \wedge y \sqsubseteq z \rightarrow x \sqsubseteq z$.

Definition: An *upper bound* of a subset E of A is an element x of A at least as great as any element of E, *i. e.* $\forall\, y \in E,\ y \sqsubseteq x$. Often, this is denoted $E \sqsubseteq x$.

Lower bounds of E are defined and denoted analogously.

Definition and Theorem: For every subset E of A there is at most one element x of A that is both an upper bound of E and a lower bound of the set of upper bounds of E, *i. e.* $E \sqsubseteq x \sqsubseteq \{\, y \mid E \sqsubseteq y\}$. Should such an element exist, it is the *least upper bound* of E and is denoted $\sqcup E$.

The greatest lower bound of E is defined analogously and denoted $\sqcap E$.

Definition: Given an increasing sequence $x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ of A, $\sqcup \{x_1, x_2, \ldots\}$, if it exists, is denoted $\sqcup x_i$ and called the *limit* of x_i.

Definition: A partial order $\sqsubseteq$ on A is *complete* if every increasing sequence has a limit.

Let V be the set of elementary data flow values such as integers and booleans. This set contains all values that could be passed between data flow operators including error values, and the **trigger** token. The set of all histories of data flow values, that is, the set of all finite and countably infinite sequences of data flow values, will be denoted V^ω. V^ω may be ordered by the *prefix* ordering:

Definition: Given two sequences x and y of V^ω, $x \sqsubseteq y$ if x is a *prefix* of y, that is, there exists a sequence z such that $x \cdot z = y$.

It is easy to verify that $\sqsubseteq$ is a complete partial order on V^ω. The least element of V^ω is the empty history, ϵ.

Recall that a history tuple is a function from a set of input or output port labels to V^ω, the set of histories. Consequently, the domain or range of a history function is the set of all functions from a set A of input or output port labels to V^ω. This set is denoted V^{ω^A}. The complete partial order $\sqsubseteq$ on V^ω can be naturally extended to V^{ω^A} by defining $X \sqsubseteq Y$, for history tuples X and Y of V^{ω^A}, if every component of X is less than the corresponding component of Y, that is, if for all a in A,

$X_a \sqsubseteq Y_a$. Again, it can be easily verified that $\sqsubseteq$ is a complete partial order on V^{ω^A}. The least element of V^{ω^A} is ϵ^A, the empty history tuple which has the empty history as each of its components.

The second requirement of fixpoint theory is that history functions be *continuous*. A function F is continuous if, for every increasing sequence $x_1 \sqsubseteq x_2 \sqsubseteq \ldots$, $F(\sqcup x_i) = \sqcup F(x_i)$. A continuous function is also *monotonic*, that is, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$. Monotonicity and continuity reflect physical properties of operator implementations.

Monotonicity implies that the more input an operator receives, the more output it will produce. This requirement reflects several implementation considerations. First, an operator cannot "withdraw" output values. Second, and perhaps most important, an operator may process its input values as they are received without the possibility that output produced in response to initial input will violate the ultimate output. If operators were not allowed this freedom and had to receive their entire input before producing any output, the potential concurrency of data flow implementations would be greatly reduced. Third, an operator cannot sense whether or not it will receive any more input. In particular, monotonicity does not allow the specification of an operator which produces the single output value true if its receives an empty input history and false otherwise.[1]

Continuity implies that no operator can produce output after receiving an infinite amount of input. An operator's response to an infinite input history must be the

---

1. This is quite different from the empty stream operator of Weng [19] which produces true if its first input token is the end of stream token.

limit of its responses to finite prefixes of that history.

Now we shall describe the derivation of the history function characterization of a data flow graph from the history functions of its operators. The syntax of graph descriptions used in Chapter 2 was chosen to emphasize that each graph operator places a relation on the histories of the graph links and ports to which it is connected. This relation is, of course, the operator's history function. When the histories of the graph input ports are fixed by a graph input history tuple, the operator history functions form a set of simultaneous equations having as their variables the labels of the links and output ports of the graph.

The result of executing a graph G with input history tuple X may be derived by use of a history function $F_{G,X}$ constructed by combining the history functions of G's operators.

Definition: Given a graph G with links and output ports labeled by elements of A and with an input history tuple X, let $F_{G,X}$ be the history function from V^{ω^A} to V^{ω^A} with the a'th component of $F_{G,X}(Z)$ determined as follows. There is, within G, *one* operator output port assigned to a. The a'th component of $F_{G,X}(Z)$ is the history of that operator output port when the operator is applied to the input history tuple consistent with the assignment of its input ports to graph ports and links and with the assignment, by history tuples X and Z, of histories to graph ports and links.

Theorem: $F_{G,X}$ is continuous.

Proof: Follows from the continuity of the history functions of the operators of G.

The result of executing G with input history tuple X is some history tuple Z such that $F_{G,X}(Z) = Z$. Only these tuples are consistent with the operator history functions.

Because $F_{G,X}$ is continuous, Scott's [16] least fixpoint operator $\mathbf{Y}$ may be used to determine the least fixpoint of the equation $F_{G,X}(Z) = Z$. The definition of $\mathbf{Y}$ follows:

Definition and Theorem: Given a continuous function F mapping a complete partially ordered set with least element $\perp$ into itself, the least solution to the equation:

$$F(x) = x$$

exists and is denoted $Y(F)$. Furthermore, letting F^n denote the function formed by composing F with itself n times:

$$Y(F) = \sqcup F^i(\perp)$$

Proof: To prove that $Y(F)$ is a fixed point, first prove that $F^i(\perp) \sqsubseteq F^{i+1}(\perp)$. To prove that $Y(F)$ is the least fixed point, first note that if $F(x) = x$ then $F^i(\perp) \sqsubseteq x$ implies $F^{i+1}(\perp) \sqsubseteq F(x) = x$.

Kahn states that $Y(F_{G,X})$ is the history tuple of the links and output ports of G resulting from the execution of G with input X. Since ϵ^A represents the history tuple that has "passed" through the links and output ports at the beginning of execution and since the passing of $F^i_{G,X}(\epsilon^A)$ implies the eventual passing of $F^{i+1}_{G,X}(\epsilon^A)$, the choice of $\sqcup F^i_{G,X}(\epsilon^A)$, or $Y(F_{G,X})$, as the result seems intuitively correct.

The history function, $\mathcal{O}[\![G]\!]$, of G is defined so that $\mathcal{O}[\![G]\!](X)$ is $Y(F_{G,X})$ restricted to the labels of the output ports of G. It is easily shown that $\mathcal{O}[\![G]\!]$ is a continuous history function. Furthermore, $\mathcal{O}[\![G]\!]$ is a complete semantic specification of G in the sense that, if H is a graph containing an operator g with the same history function as G, the graph $H[g/G]$ obtained from H by substituting G for each occurrence of G has the same history function as H. Consequently, in deriving the history function of a graph, subgraphs and operators may be treated alike. Subgraphs do not have to be expanded into their operator implementations.

In the last section of this chapter, the least fixpoint derivation of a data flow graph will be given. Readers desiring more complete proofs of the theorems stated in this section should consult the work of Kahn [12] and Scott [16].

3.2 The Semantic Specification of the Data Flow Operators

All ADFL operators and constants have an interpretation. The interpretation, $\mathcal{I}[\![oper]\!]$, of an operator *oper* is a function from V^m to V^n. $\mathcal{I}[\![oper]\!]$ is the usual arithmetic or Boolean function associated with *oper*. For example:

$$\mathcal{I}[\![+]\!](x, y) = x + y$$
$$\mathcal{I}[\![\wedge]\!](x, y) = x \wedge y$$

$\mathcal{I}[\![oper]\!]$ is assumed to map "inappropriate" input tuples, such as those containing values of an unexpected type, into some appropriate tuple of output values.

The history function, $\mathcal{O}[\![oper]\!]$, of the data flow operator *oper* maps *m*-tuples of input histories into *n*-tuples of output histories. The data flow operator receives a sequence of input *m*-tuples and computes the sequence of *n*-tuples resulting from the application of $\mathcal{I}[\![oper]\!]$ to each input *m*-tuple. Furthermore, the firing rule of the data flow operator is *strict*. The operator will not fire without a complete tuple of inputs.

$$\mathcal{O}[\![oper]\!](X) = \epsilon^n, \text{ if } \exists\, i \ni X_i = \epsilon$$
$$\mathcal{O}[\![oper]\!](x \cdot X) = \mathcal{I}[\![oper]\!](x) \cdot \mathcal{O}[\![oper]\!](X), \text{ if } x \in V^m$$

Because $\mathcal{O}[\![oper]\!]$ must be continuous, it suffices to define $\mathcal{O}[\![oper]\!]$ only on finite input history tuples.

The interpretation, $\mathcal{I}[\![const]\!]$, of a ADFL constant *const* is an element of **V**. The history function, $\mathcal{O}[\![const]\!]$, of the data flow constant operator *const* maps V^ω into V^ω. Data flow graphs are constructed so that constant operators receive only **trigger** input values. An output of value $\mathcal{I}[\![const]\!]$ is produced for every **trigger** input received.

$$\mathcal{O}[\![const]\!](\epsilon) = \epsilon$$
$$\mathcal{O}[\![const]\!](\textbf{trigger} \cdot X) = \mathcal{I}[\![const]\!] \cdot \mathcal{O}[\![const]\!](X)$$

Recall the firing rules of the data flow gate operators. The gate operators are not strict. They absorb values from selected input ports. The history function of the T gate, respectively F gate, maps V^{ω^2} into V^{ω}. When a true, respectively false, control value is received, the data value is absorbed and passed through the output port. When any other control value is received, the data value is absorbed and no output is produced.

$$\bigcirc[\![\text{T gate}]\!](\epsilon, Y) = \epsilon$$
$$\bigcirc[\![\text{T gate}]\!](x \cdot X, \epsilon) = \epsilon$$
$$\bigcirc[\![\text{T gate}]\!](\text{true} \cdot X, y \cdot Y) = y \cdot \bigcirc[\![\text{T gate}]\!](X, Y)$$
$$\bigcirc[\![\text{T gate}]\!](x \cdot X, y \cdot Y) = \bigcirc[\![\text{T gate}]\!](X, Y), \text{ if } x \neq \text{true}$$

$$\bigcirc[\![\text{F gate}]\!](\epsilon, Y) = \epsilon$$
$$\bigcirc[\![\text{F gate}]\!](x \cdot X, \epsilon) = \epsilon$$
$$\bigcirc[\![\text{F gate}]\!](\text{false} \cdot X, y \cdot Y) = y \cdot \bigcirc[\![\text{T gate}]\!](X, Y)$$
$$\bigcirc[\![\text{F gate}]\!](x \cdot X, y \cdot Y) = \bigcirc[\![\text{T gate}]\!](X, Y), \text{ if } x \neq \text{false}$$

The history function $\bigcirc[\![\text{M gate}]\!]$ maps V^{ω^3} into V^{ω}. The control value selects which data value is passed to the output port. If a non-Boolean control value is received, no data value is absorbed and err is output. The FM gate is a M gate with a built-in initial false control value.

$$\bigcirc[\![\text{M gate}]\!](\epsilon, Y, Z) = \epsilon$$
$$\bigcirc[\![\text{M gate}]\!](\text{true} \cdot X, \epsilon, Z) = \epsilon$$
$$\bigcirc[\![\text{M gate}]\!](\text{true} \cdot X, y \cdot Y, Z) = y \cdot \bigcirc[\![\text{M gate}]\!](X, Y, Z)$$
$$\bigcirc[\![\text{M gate}]\!](\text{false} \cdot X, Y, \epsilon) = \epsilon$$
$$\bigcirc[\![\text{M gate}]\!](\text{false} \cdot X, Y, z \cdot Z) = z \cdot \bigcirc[\![\text{M gate}]\!](X, Y, Z)$$
$$\bigcirc[\![\text{M gate}]\!](x \cdot X, Y, Z) = \text{err} \cdot \bigcirc[\![\text{M gate}]\!](X, Y, Z), \text{ if } x \notin \{\text{true}, \text{false}\}$$

$$\bigcirc[\![\text{FM gate}]\!](X, Y, \epsilon) = \epsilon$$
$$\bigcirc[\![\text{FM gate}]\!](X, Y, z \cdot Z) = z \cdot \bigcirc[\![\text{FM gate}]\!](X, Y, Z)$$

The history function $\bigcirc[\![\text{FS gate}]\!]$ maps V^{ω^2} into V^{ω}. When a false control value is received, the FS gate passes its data value and sets an internal register to

that value. When a **true** control value is received, the **FS gate** absorbs no data value but outputs the value contained in its register. The **FS gate** has an initial built-in **false** control value. The control value of the **FS gate** is the **iter?** value of an iteration body and, consequently, must be either **true** or **false**.

$\mathcal{O}[\![\text{FS gate}]\!](X, \epsilon) = \epsilon$
$\mathcal{O}[\![\text{FS gate}]\!](X, y \cdot Y) = y \cdot S_y(X, Y)$
$S_z(\epsilon, Y) = \epsilon$
$S_z(\text{true} \cdot X, Y) = z \cdot S_z(X, Y)$
$S_z(\text{false} \cdot X, \epsilon) = \epsilon$
$S_z(\text{false} \cdot X, y \cdot Y) = y \cdot S_y(X, Y)$

Compare the history function specification of the **IC gate** with the table specification of its firing rules given on page 23.

$\mathcal{O}[\![\text{IC gate}]\!](\epsilon, Y, Z) = (\epsilon, \epsilon, \epsilon)$
$\mathcal{O}[\![\text{IC gate}]\!](\text{true} \cdot X, \epsilon, Z) = (\epsilon, \epsilon, \epsilon)$
$\mathcal{O}[\![\text{IC gate}]\!](\text{true} \cdot X, \text{true} \cdot Y, Z) = (\text{true}, \text{true}, \epsilon) \cdot \mathcal{O}[\![\text{IC gate}]\!](X, Y, Z)$
$\mathcal{O}[\![\text{IC gate}]\!](\text{true} \cdot X, \text{false} \cdot Y, Z) = (\text{false}, \epsilon, \text{true}) \cdot \mathcal{O}[\![\text{IC gate}]\!](X, Y, Z)$
$\mathcal{O}[\![\text{IC gate}]\!](\text{false} \cdot X, Y, \epsilon) = (\epsilon, \epsilon, \epsilon)$
$\mathcal{O}[\![\text{IC gate}]\!](\text{false} \cdot X, Y, \text{true} \cdot Z) = (\text{true}, \text{false}, \epsilon) \cdot \mathcal{O}[\![\text{IC gate}]\!](X, Y, Z)$
$\mathcal{O}[\![\text{IC gate}]\!](\text{false} \cdot X, Y, \text{false} \cdot Z) = (\text{false}, \epsilon, \text{false}) \cdot \mathcal{O}[\![\text{IC gate}]\!](X, Y, Z)$
$\mathcal{O}[\![\text{IC gate}]\!](x \cdot X, Y, Z) = (\text{false}, \epsilon, \text{err}) \cdot \mathcal{O}[\![\text{IC gate}]\!](X, Y, Z)$, if $x \notin \{\text{true}, \text{false}\}$

The **E gate**, the only remaining gate, passes its data value when it receives a Boolean control value, and absorbs no data value and produces **err** when it receives a non-Boolean control value.

$\mathcal{O}[\![\text{E gate}]\!](\epsilon, Y) = \epsilon$
$\mathcal{O}[\![\text{E gate}]\!](x \cdot X, \epsilon) = \epsilon$, if $x \in \{\text{true}, \text{false}\}$
$\mathcal{O}[\![\text{E gate}]\!](x \cdot X, y \cdot Y) = y \cdot \mathcal{O}[\![\text{E gate}]\!](X, Y)$, if $x \in \{\text{true}, \text{false}\}$
$\mathcal{O}[\![\text{E gate}]\!](x \cdot X, Y) = \mathcal{O}[\![\text{E gate}]\!](X, Y)$, if $x \notin \{\text{true}, \text{false}\}$

The history function specifications of the data flow operators of Chapter 2 completed, the operational semantics $\mathcal{O} \circ \mathcal{F}[\![exp]\!]$ of an ADFL expression *exp* may be

obtained by using the translation algorithm to construct the data flow graph $\mathcal{T}[\![exp]\!]$ and using the operator history function specifications and Kahn's fixpoint theory to derive $\mathcal{O}\circ\mathcal{T}[\![exp]\!]$. This method of deriving operational semantics is illustrated in the following section.

3.3 Operational Semantics for an ADFL Expression

In this section the operational semantics of the iteration expression "let *id*1, ..., *idn* = *exp* in *iteration*" is derived assuming the operational characteristics of its component initialization expression and iteration body. For convenience, let *G*, *Ginit*, and *Giter*, denote the data flow graphs of, respectively, the iteration expression, the initialization expression, and the iteration body. Recall from Chapter 2 the *operators* list of the graph description of *G*.

Ginit inputs: $(a \in Ginit)\ a \rightarrow a$
outputs: $(i \in 1\ ..\ n)\ i \rightarrow \alpha_i$
$(i \in 1\ ..\ n)$ FM gate inputs: $\gamma_{iter?} \rightarrow 1, \gamma_i \rightarrow 2, \alpha_i \rightarrow 3$
outputs: $1 \rightarrow \beta_{idI}$
$(a \in IN(Giter) - \{id1, ..., idn\})$ FS gate inputs: $\gamma_{iter?} \rightarrow 1, a \rightarrow 2$
outputs: $1 \rightarrow \beta_a$
Giter inputs: $(a \in IN(Giter))\ \beta_a \rightarrow a$
outputs: iter? $\rightarrow \gamma_{iter?}$, $(i \in 1\ ..\ n)\ Ii \rightarrow \gamma_i$, $(Ri \in ROUT(Giter))\ Ri \rightarrow i$

We assume the history functions $\mathcal{O}[\![Ginit]\!]$ and $\mathcal{O}[\![Giter]\!]$ have been derived, recursively, using fixpoint theory.

$\mathcal{O}[\![G]\!](X)$ is found by deriving the least fixed point of $F_{G,X}$. In Section 3.1, $F_{G,X}$ was defined as a history function from V^{ω^A} to V^{ω^A} where *A* contains the labels of the links and output ports of *G*. From the graph description we see that *A* contains:

$(i \in 1\ ..\ n)\ \alpha_i$, $(a \in IN(Giter))\ \beta_a$, $\gamma_{iter?}$, $(i \in 1\ ..\ n)\ \gamma_i$, $(Ri \in ROUT(Giter))\ i$

Given Z of V^{ω^A}, let Z_{Giter} denote, in a slight abuse of notation, the tuple mapping

IN(*Giter*) into V, and let X_{Ginit} denote the tuple mapping IN(*Ginit*) into V, such that:

$$Z_{Giter}(a) = Z(\beta_a)$$
$$X_{Ginit}(a) = X(a)$$

Z_{Giter} is defined to reflect the assignment of input port *a* of *Giter* to link β_a of *G*; and X_{Ginit}, the assignment of input port *a* of *Ginit* to input port *a* of *G*. $F_{G,X}(Z)$ is the element of $V^{(\omega)^A}$ such that:

$$F_{G,X}(Z)(\alpha_i) = \bigcirc[\![Ginit]\!](X_{Ginit})(i), \text{ if } i \in 1 \,..\, n$$
$$F_{G,X}(Z)(\beta_{id1}) = \bigcirc[\![\text{FM gate}]\!](Z(\gamma_{iter?}), Z(\gamma_i), Z(\alpha_i)), \text{ if } i \in 1 \,..\, n$$
$$F_{G,X}(Z)(\beta_a) = \bigcirc[\![\text{FS gate}]\!](Z(\gamma_{iter?}), X(a)), \text{ if } a \in \text{IN}(Giter) - \{id1, ..., idn\}$$
$$F_{G,X}(Z)(\gamma_{iter?}) = \bigcirc[\![Giter]\!](Z_{Giter})(\text{iter?})$$
$$F_{G,X}(Z)(\gamma_i) = \bigcirc[\![Giter]\!](Z_{Giter})(\text{I}i), \text{ if } i \in 1 \,..\, n$$
$$F_{G,X}(Z)(i) = \bigcirc[\![Giter]\!](Z_{Giter})(\text{R}i), \text{ if R}i \in \text{ROUT}(Giter)$$

Suppose all input history components of *X* contain a single value. That is, suppose *X* represents a single set of input values to *G*. Further suppose that *G*, given input *X*, iterates *m*+1 times before producing its output tuple. Let V_0 be the *n*-tuple produced by the initialization expression subgraph, *Ginit*, and initially bound to the iteration variables, *id*1, ..., *idn*. Let V_1, ..., V_m be the *n*-tuples produced by the first *m* iterations of the iteration subgraph, *Giter*, and let *W* be the ultimate, non-iter, output tuple produced on the final iteration. The formal relation between these tuples and the history functions $\bigcirc[\![Ginit]\!]$ and $\bigcirc[\![Giter]\!]$ follows.

Since V_0 is produced by *Ginit*

$$V_0 = \bigcirc[\![Ginit]\!](X_{Ginit})$$

On the *j*'th iteration, the input tuple V_{j-1} is received at input ports of *Giter* labeled by the iteration variables. Other input ports receive values contained in the graph input tuple, *X*. Let VX_{j-1} represent this input tuple.

$VX_{j-1}(idi) = V_{j-1}(i)$, if $i \in 1 .. n$
$VX_{j-1}(a) = X(a)$, if $a \in$ IN(*Giter*) - $\{id1, ..., idn\}$

At the end of the j'th iteration of *Giter*, for j not greater than m; the tuple V_j is produced on the I output ports of *Giter*; **true**, on the **iter?** output port; and **no values**, on the R output ports. Consequently:

$()[\![Giter]\!](VX_0 \cdot ... \cdot VX_{j-1})$(**iter?**) = **true**^{j}
$()[\![Giter]\!](VX_0 \cdot ... \cdot VX_{j-1})(Ii) = V_0(i) \cdot ... \cdot V_j(i)$, if $i \in 1 .. n$
$()[\![Giter]\!](VX_0 \cdot ... \cdot VX_{j-1})(Ri) = \epsilon$, if $Ri \in$ ROUT(*Giter*)

Where **true**^{j} is the sequence of j **true** values. At the end of the last, m+1'st, iteration of *Giter*; the tuple W is produced on the R output ports of *Giter*; **false**, on the **iter?** output port; and no values, on the I output ports. Consequently:

$()[\![Giter]\!](VX_0 \cdot ... \cdot VX_m)$(**iter?**) = **true**^{m}**·false**
$()[\![Giter]\!](VX_0 \cdot ... \cdot VX_m)(Ii) = V_0(i) \cdot ... \cdot V_m(i)$, if $i \in 1 .. n$
$()[\![Giter]\!](VX_0 \cdot ... \cdot VX_m)(Ri) = W(i)$, if $Ri \in$ ROUT(*Giter*)

Using this history function specification of *Giter* and *Ginit*, the reader may verify that for the least fixed point $Y(F_{G,X})$, or $\sqcup F^j_{G,X}(\epsilon^A)$, of $F_{G,X}$ is the tuple mapping A, the labels of the links and output ports of G, into V such that:

$Y(F_{G,X})(\alpha_i) = V_0(i)$, if $i \in 1 .. n$
$Y(F_{G,X})(\beta_{idi}) = V_0(i) \cdot ... \cdot V_m(i)$, if $i \in 1 .. n$
$Y(F_{G,X})(\beta_a) = X(a)^{m+1}$, if $a \in$ IN(*Giter*) - $\{id1, ..., idn\}$
$Y(F_{G,X})(\gamma_{iter?}) =$ **true**^{m}**·false**
$Y(F_{G,X})(\gamma_i) = V_1(i) \cdot ... \cdot V_m(i)$, if $i \in 1 .. n$
$Y(F_{G,X})(i) = W(i)$, if $Ri \in$ ROUT(*Giter*)

Consequently, $()[\![G]\!](X)$ is W, $Y(F_{G,X})$ restricted to the output port labels of G. As expected, W is the output tuple produced by the final iteration. Note that **false** was produced as the iter? value on the final iteration, thus resetting the **FM gate** and **FS gate's** for a new set of inputs. This example derivation demonstrates how the **data** flow graph implementation of the iteration expression satisfies its intended **function.**

The operational semantics of any ADFL expression may be derived similarly. First, the expression is translated into a data flow graph. The operational semantics of the expression is the history function of its graph. The history function of the graph is obtained by recursively using Kahn's theory to obtain the history functions of the subgraphs corresponding to the syntactic components of the expression. The basis of the recursion is the history function characterizations of the elementary data flow operators.

4. Conclusions and Suggestions for Future Research

The operational semantics of ADFL, an applicative data flow language with an iteration construct resembling tail recursion, have been expressed as a two step process. In the first step, the application of the translation algorithm $\mathcal{T}$ to an ADFL expression yields its data flow graph implementation. In the second step, the application of the semantic function $\mathcal{O}$ to the graph yields its semantic characterization. The graph is an explicit representation of the concurrency possible in evaluation of the expression. It is an interconnection of data flow operators, corresponding to ADFL operators, which communicate values to each other through input and output ports. In conventional sequential control flow evaluation, operators are performed in a pre-ordained sequential order. In data flow evaluation, operators are performed as soon as their arguments are available.

The translation algorithm $\mathcal{T}$ is recursive. The graph of an expression is constructed from subgraphs implementing its syntactic subcomponents. The graph has an input port for each free variable of the expression and an output port for each value returned by the expression. Data flow graphs are specified with a graph assembly language well-suited for describing $\mathcal{T}$.

When expressions are evaluated under sequential control flow, execution exceptions are often handled by interrupts. However, in ADFL an error-handling scheme more appropriate to both the concurrency of data flow and the value-orientation of the language is used. Special error values are returned when exceptions occur. Conditional and iteration expressions are implemented with special gates designed to be consistent with this error-handling philosophy.

In the second step in obtaining the operational semantics of an ADFL expression, the application of the semantic function $\mathcal{O}$ to the graph of the expression yields its semantic characterization. The result of executing a graph is characterized by a history function mapping a tuple of input histories into a unique tuple of output histories. The history function of a graph is derived by use of Kahn's fixpoint theory of communicating interconnecting processes. Here, the processes are the data flow operators. Thus, the operational semantics of an ADFL expression is obtained by application of $\mathcal{O} \circ \mathcal{T}$. $\mathcal{O} \circ \mathcal{T}$ maps an ADFL expression through its data flow graph implementation to its history function characterization.

There are three avenues for extending this research. First, the language may be extended. Second, the operational characterization of data flow graphs may be modified to more closely correspond to execution on specific data flow machines. And third, an alternative semantic characterization of data flow languages may be given and proven consistent with this operational characterization.

The most obvious language extension is the addition of procedures. Procedures may be implemented at the data flow graph level with an apply operation which receives a data flow graph on one input port and values to which the graph is to be applied on its remaining input ports. Since Kahn's theory can be extended to include recursive graphs, it is easy to characterize the operational semantics of such a data flow language.

Another language extension is the incorporation of the determinate stream operators of Weng [19]. A stream is a list whose elements are generated over time. Stream operators process these lists one element at a time. Consequently, the concurrency of data flow program execution is increased by allowing a data flow

operator to process elements of an input stream value while elements of an output stream value are being generated. Determinate stream operators are naturally characterized by history functions, and, thus, Kahn's fixpoint theory may be used to define the operational semantics of a language with determinate stream operators.

A non-determinate stream operator, merge, has been used by Arvind, Gostelow, and Plouffe [4] and Dennis [8] in data flow implementations of real-time systems, such as resource allocators and airline reservation systems. The merge operator accepts two input streams and merges them into one output stream. The output stream may be one of several interleavings of the input streams. It is difficult to extend Kahn's fixpoint theory to non-determinate computation. Brock and Ackerman [6] have shown that arbitrary non-determinate data flow graphs cannot be operationally characterized by the natural extension of history functions, a mapping from tuples of input histories to sets of tuples of output histories, while Kosinski [13] has described an operational semantics of non-determinate data flow graphs in which each data flow value is "tagged" with the non-determinate "choices" leading to its generation. Kosinski's theory seems unnecessarily complicated since non-determinate computations may be simulated without tagging values. Consequently, a simpler characterization of non-determinate data flow computation may exist. An alternative area of research is finding a non-determinate data flow language which restricts the use of merge operators so that graphs have a simple operational characterization.

The second avenue of extending this research is the operational characterization of data flow computation on specific machines. Kahn's theory assumes that the links of data flow graphs are unbounded FIFO queues; however, in the data flow machine design of Dennis and Misunas [9], the links are one-place

buffers. If operators are allowed to "write over" buffered values, graph computation is non-determinate. Presently, Montz [14] is investigating the use of *acknowledge* signals to control operator firings. In this scheme, whenever an output and input operator are connected by a link; a second acknowledge link is placed, in the opposite direction, between the operators. The output operator will not place a value on the data link until it has received an acknowledge value, and the input operator generates an acknowledge value whenever it removes a data value. Semantically, graphs constructed with this acknowledge protocol may be considered to contain links implemented by unbounded FIFO queues; although, in actual execution, only one place of the queues will ever be used.

The third avenue is proving that the operational semantics of ADFL are consistent with a more abstract semantic characterization. The denotational semantics [18] of a language are given by defining a direct mapping of syntactic components to suitable abstract objects. For example, procedures may be mapped into functions without regard to details of implementation or execution. Scott's [16] theory provides the theoretical basis for defining iterative computation and for constructing abstract objects for syntatic components. Since ADFL is applicative, the sole effect of evaluation is to return a tuple of values dependent solely on the values bound to the free identifiers of the evaluated expression. Consequently, any expression of ADFL may be denotationally characterized by a function mapping each *environment*, association of identifiers and values, into the tuple of values returned when the expression is evaluated within that environment. The denotational semantics of ADFL have a simple, elegant statement. Further research of this author will prove that the operational and denotational semantics of ADFL are consistent.

Bibliography

[1] Ackerman, W. B., *A Structure Memory for Data Flow Computers*, Laboratory for Computer Science (TR-186), MIT, Cambridge, Massachusetts, August 1977.

[2] Ackerman, W. B., and J. B. Dennis, "VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual", Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, In Preparation.

[3] Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time", *Information Processing 77: Proceedings of IFIP Congress 77* (B. Gilchrist, Ed.), August 1977, 849-853.

[4] Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors and Dataflow", *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, *Operating Systems Review* 11, 5(November 1977), 159-169.

[5] Arvind, K. P. Gostelow, and W. Plouffe, *The (Preliminary) Id Report: An Asynchronous Programming Language and Computing Machine*, Department of Information and Computer Science (TR 114), University of California - Irvine, Irvine, California, May 1978.

[6] Brock, J. D., and W. B. Ackerman, "An Anomaly in the Specifications of Nondeterminate Packet Systems", Computation Structures Group (Note 33-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1978.

[7] Dennis, J. B., "First Version of a Data Flow Procedure Language", *Programming Symposium: Proceedings, Colleque sur la Programmation* (B. Robinet, Ed.), *Lecture Notes in Computer Science* 19, 362-376.

[8] Dennis, J. B., "A Language Design for Structured Concurrency", *Proceedings of a DoD Sponsored Workshop* (J. H. Williams, and D. A. Fisher, Eds.), *Lecture Notes in Computer Science* 54, October 1976.

[9] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", *The Second Annual Symposium on Computer Architecture: Conference Proceedings*, January 1975, 126-132.

[10] Ellis, D. J, *Formal Specifications for Packet Communication Systems*, Laboratory for Computer Science (TR-189), MIT, Cambridge, Massachusetts, November 1977.

[11] Hoare, C. A. R., "Communicating Sequential Processes", *Communications of the ACM* 21, 8(August 1978), 666-677.

[12] Kahn, G., "The Semantics of a Simple Language for Parallel Programming", *Information Processing 74: Proceedings of the IFIP Congress 74*, August 1974, 471-475.

[13] Kosinski, P. R., "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, January 1978, 214-221.

[14] Montz, L., *Safety and Optimization Transformations for Data Flow Programs*, S. M. Thesis in preparation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, expected January 1979.

[15] Patil, S. S., "Closure Properties of Interconnections of Determinate Systems", *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970, 107-116.

[16] Scott, D. S., "Data Types as Lattices", *SIAM Journal of Computing* 5, 3(September 1976), 522-587.

[17] Steele, G. L., *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)*, Artificial Intelligence Laboratory (AI-TR-474), MIT, Cambridge, Massachusetts, 1978.

[18] Tennent, R. D., "The Denotational Semantics of Programming Languages", *Communications of the ACM* 19, 8(August 1976), 437-453.

[19] Weng, K.-S., *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science (TM-68), MIT, Cambridge, Massachusetts, October 1975.