LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Consistent Semantics for a Data Flow Language

Computation Structures Group Memo 172-1
June 1980

J. Dean Brock

This paper appeared in the Proceedings of the Ninth International Symposium on Mathematical Foundations of Computer Science, Rydzyna, Poland, September 1980.

This research was supported by the National Science Foundation under contract 7915255 and the Department of Energy under contract DE-AC02-79ER10473.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Consistent Semantics for a Data Flow Language*

J. Dean Brock

Laboratory for Computer Science

Massachusetts Institute of Technology

Cambridge, Massachusetts 02139 USA

1. Introduction

Recent attempts to design programming languages for specifying concurrent computation, and, consequently, recent attempts to semantically characterize concurrent computation, have been "hardware-driven." Programming languages have made the transition from modeling a single serial von Neumann process and its memory; to modeling several processes sharing a memory; to modeling several communicating processes, each with its own memory. Concurrent programming concepts, such as Hoare's [9] *communicating sequential processes* and Brinch Hansen's [4] *distributed processes*, may be forgiven their semantic complexity, since they introduce abstraction to the very important application area of real-time systems presently dominated by *ad hoc* machine language programming, and since most real-time systems are inherently non-determinate and time-dependent and therefore beyond straightforward semantic description. However, there are many semantically "simple" application areas, such as numerical simulation, in which any performance benefits that could be gained using concurrency are overwhelmed by the programming cost of partitioning tasks among several communicating processes. This partitioning task would be immense on envisioned computer architectures exploiting the advantages of VLSI technology by incorporating thousands of very small processing elements. Consequently, it is imperative that models of concurrent computation other than variations of the von Neumann model be investigated. The data flow model of computation [6] is one such model.

A data flow program may be translated into a data flow graph in which the grain of concurrency appears at the level of elementary program operations instead of procedures consisting of several program statements. Furthermore, data flow programming languages are applicative languages, and, consequently, share with applicative languages elegance of semantic characterization and ease of program verification.

In this paper we will examine the data flow model of computation and define ADFL, an Applicative Data Flow Language. Additionally, the denotational and operational semantics of ADFL will be given and shown to be consistent. Scott's [11] fixpoint theory will be used to specify the denotational semantics. The operational semantics are given by a two step process. One step corresponds to the translation of programs into data flow graphs, while the other corresponds to the execution of the resulting graphs. The result of graph execution is derived using Kahn's [10] fixpoint theory of communicating processes.

The denotational and operational semantics of ADFL are not equivalent. The denotational semantics specify that expression evaluation must terminate to yield results and that, if expression evaluation terminates, all

---

*This research was supported by the National Science Foundation under contract 7915255-MCS and the Department of Energy under contract DE-AC02-79ER10473.

subexpression evaluations terminate. However, in data flow, and many other models of concurrent computation, a computation may produce results even if some internal computations do not terminate. The characterization of such computations contributes much to the complexity of the operational semantics of ADFL. Consequently, the simpler denotational semantics are the more useful in tasks such as program verification. The proof of consistency assures those using the simpler semantics that the two semantic theories agree on all "denotationally" terminating expression evaluations.

1.1 The Data Flow Model of Computation

The program schema of the data flow model of computation is the *data flow graph*, a directed graph whose nodes are called *operators*. Each operator has labeled input and output ports, and graph links are directed from operator input ports to operator output ports. Graphs, like operators, have input and output ports. The unlinked operator ports within a graph are the ports of the graph itself.

The execution of a data flow graph can be interpreted within Kahn's [10] fixpoint theory by viewing the operators as parallel programs and the links as channels for program (operator) communication. Because graph operators correspond to elementary program operators, they are exceeding simple parallel "programs." For most operators, execution consists of a repetition of *firings*. Operators are *enabled* for firing by the presence of input values. In firing, an operator accepts values at its input ports and produces results at its output ports. The following program of Kahn's parallel programming language implements the data flow + operator:

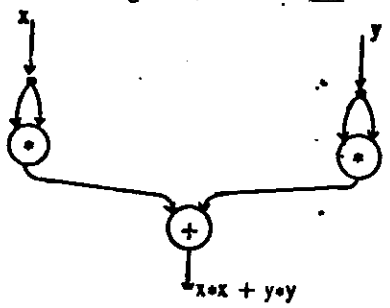
```
Process plus(integer in I1, I2; integer out O);
 Repeat Begin Integer T;
  Comment: the order of the wait invocations is irrelevant;
  T := wait(I1) + wait(I2);
  send T on O;
 End;
```

The data flow graph and parse tree representation of elementary expressions are very similar. The graph representing an application of + to two arguments is formed by linking the output ports of the graphs computing the two arguments to the input ports of a + operator. The data flow graph for the simple expression:

```
x*x + y*y
```

is illustrated in Figure 1. Note that the graph input ports are labeled by the free variables of the expression. If a free variable occurs more than once, a copy operator (represented in the figure by a solid black dot) is used to distribute the

Figure 1. A Sample Data Flow Program



variable. The data flow graph implementations of other commonly used programming constructs will be discussed in Section 3.

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 [8] and Arvind and Gostelow [2]. 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 lament.

The "First Version of a Data Flow Language" by Dennis [6] was a rudimentary ALGOL-like language. Most data flow language are statement-oriented languages given an applicative flavor by imposing the *single-assignment* rule: Programs are syntactically restricted to guarantee that each variable would be assigned only one value during the program's execution. The languages of Weng [13] and Arvind, Gostelow, and Plouffe [3], in addition to having the expressive power of ALGOL, facilitate the programming of networks of communicating processes, such as co-routines and operating systems.

1.2 ADFL - An Applicative Data Flow Language

ADFL, Applicative Data Flow Language, is a simplification of VAL, the Value-oriented Algorithmic Language developed by Ackerman and Dennis [1]. A BNF specification of the syntax of ADFL follows:

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

iterbody ::= *exp* | iter(*exp*) | let *idlist* = *exp* in *iterbody* end |
if *exp* then *iterbody* else *iterbody* end

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 end

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** *iterbody* **end**

is accomplished by first binding the *iteration identifiers*, the elements of *idlist*, to the values of *exp*. Note from the BNF specification of *iterbody*, 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 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 end
end
```

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. A detailed specification of the class of error values and the results of operator application to error values is given in the documentation of VAL [1].

2. 8: The Denotational Semantics of ADFL

ADFL has a simple denotational characterization, similar to those given by Scott [11] and by Tennent [12] for other applicative languages. Before proceeding, we briefly review some of the notations and concepts of fixpoint theory.

Notation: Given a set A with partial ordering $\sqsubseteq$, the least upper bound of a subset E of A is denoted $\sqcap E$, and the *limit*, $\sqcap \{x_1, x_2, ...\}$, of an increasing sequence $x_1 \sqsubseteq x_2 \sqsubseteq ...$ of A is denoted $\sqcap x_i$.

Definition: A *domain* is a partially ordered set A with a least element, usually denoted $\perp$, such that every increasing sequence of A has a limit.

Definitions: A function F from domain A to domain B is *continuous* if, for every increasing sequence $x_1 \sqsubseteq x_2 \sqsubseteq ...$, $F(\sqcap x_i) = \sqcap F(x_i)$. Every continuous function F is also *monotonic*, that is, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$.

Definition and Theorem: Given domains A and B, the *product* domain $A \times B$, populated by the elements of the Cartesian product of A and B and ordered so that $(x_1, y_1) \sqsubseteq (x_2, y_2)$ if and only if $x_1 \sqsubseteq x_2$ and $y_1 \sqsubseteq y_2$, and the *function* domain $A \rightarrow B$, populated by the continuous functions from A to B and ordered so that $F \sqsubseteq G$ if and only if $F(x) \sqsubseteq G(x)$ for all elements x of A, are domains.

Definition and Theorem: Given a continuous function F of $A \rightarrow A$, the *least fixpoint* (solution) to the equation

$$F(x) = x$$

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

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

Let V be the set of all values of ADFL, V^* be the set of all tuples of values, and $V^*_\perp$ be the *discrete* value domain formed by adjoining to V^* a least element $\perp$. $V^*_\perp$ is ordered discretly, that is, for all elements x and y of $V^*_\perp$, $x \sqsubseteq y$ if and only if $\perp = x$ or $x = y$. The five semantic categories of ADFL, constants, operators, identifiers, expressions, and iteration bodies, will be denoted, respectively, Const, Oper, Id, Exp, and Iterbody. The semantic function $\mathcal{I}$ maps ADFL constants and operators into their interpretations. The interpretation $\mathcal{I}[\![const]\!]$ of a constant *const* is an element of V^*. $\mathcal{I}[\![oper]\!]$ is the usual arithmetic or Boolean function associated with *oper*, extended to $V^*_\perp$ by defining applications of $\mathcal{I}[\![oper]\!]$ to tuples inappropriate in type to map into special error values. For example:

$\mathcal{I}[\![+]\!](x, y) = x + y$, if x and y are integer values
$\mathcal{I}[\![\wedge]\!](x, y) = x \wedge y$, if x and y are boolean values

A complete specification of $\mathcal{I}$ will not be given here. However, the denotational interpretation of operators is required to be *strict* in the following sense: $\mathcal{I}[\![oper]\!](x) = \perp$ if and only if $x = \perp$. Because $\perp$ will correspond to the result of a non-terminating computation, this strictness requirement insures that an application of *oper* to an expression terminates if and only if the expression terminates. In addition, this requirement insures that $\mathcal{I}[\![oper]\!]$ is continuous.

Because ADFL is applicative, its expressions may be denotationally characterized by a function mapping each *environment*, association of identifiers and values, into the tuple of values returned by expression evaluation within that environment. Let U, the environments of ADFL, be $Id \rightarrow V$, the continuous functions from Id to V. The semantic function $\mathcal{E}$ is the expression evaluation function of ADFL. $\mathcal{E}$, a member of $Exp \rightarrow U \rightarrow V^*_\perp$, maps (in curried notation) expressions and environments into tuples of values. In the specification of $\mathcal{E}$, expressions which a compiler could declare "invalid" are ignored. Such expressions are invalid either because they contain instances in which an unbound (uninitialized) identifier could be evaluated or instances in which an operator could be applied to an expression of inappropriate arity.

The specification of $\mathcal{E}$ for expressions without iteration is trivial. Evaluation of an identifier yields the value of the identifier within the current environment, evaluation of a constant yields $\mathcal{I}[\![const]\!]$, and evaluation of an operator application is accomplished by applying $\mathcal{I}[\![oper]\!]$ to the values of the argument expression.

$\mathcal{E}[\![id]\!]\rho = \rho[\![id]\!]$
$\mathcal{E}[\![const]\!]\rho = \mathcal{I}[\![const]\!]$
$\mathcal{E}[\![oper(exp)]\!]\rho = \mathcal{I}[\![oper]\!](\mathcal{E}[\![exp]\!]\rho)$

Let $\|_s$ denote the strict concatenation operator over elements of $V^*_\perp$. That is, for tuples x and y of V^*, $x \|_s y$ is $x \| y$, the concatenation of x and y, and $x \|_s \perp$ and $\perp \|_s y$ are both $\perp$. Using the strict concatenation operator, we define the value of a tuple of expressions to be $\perp$ if one of its component expressions is $\perp$.

$\mathcal{E}[\![exp_1, exp_2]\!]\rho = \mathcal{E}[\![exp_1]\!]\rho \|_s \mathcal{E}[\![exp_2]\!]\rho$

The updated environment resulting from binding the values of a tuple x to successive identifiers of a list *idlist* in environment ρ is denoted $\rho[idlist/x]$. Our denotational specification requires the binding expression and the conditional expression to be strict. The *precedes* function $\Rightarrow$ and the *condition* function $\rightarrow$ enforce strictness.

$\perp \Rightarrow y = \perp$
$x \Rightarrow y = y$, if $x \neq \perp$

$\mathcal{E}[\![\text{let } idlist = exp_1 \text{ in } exp_2 \text{ end}]\!]\rho = \mathcal{E}[\![exp_1]\!]\rho \Rightarrow \mathcal{E}[\![exp_2]\!]\rho[idlist/\mathcal{E}[\![exp_1]\!]\rho]$

$\perp \rightarrow x, y = \perp$
true $\rightarrow x, y = x$
false $\rightarrow x, y = y$
$z \rightarrow x, y =$ ... some error value ..., if $z \notin \{\perp, \text{true}, \text{false}\}$
$\mathcal{E}[\![\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \text{ end}]\!]\rho = \mathcal{E}[\![exp_1]\!]\rho \rightarrow \mathcal{E}[\![exp_2]\!]\rho, \mathcal{E}[\![exp_3]\!]\rho$

Evaluation of the iteration expression "let *idlist* = *exp* in *iterbody* end" could be specified by considering *iterbody* to be a recursive procedure, with name iter and parameters *idlist*. However, for the proof of consistency, it is more convenient to view the iteration body as returning a tuple with a tag indicating whether or not iteration is to be continued or terminated. To do so, we extend tuples from their mathematical foundation as functions whose domain is a subset of the integers to functions with arbitrary domains. Note that environments are such tuples. The tag of the tuple x returned by the iteration body is appropriately denoted x_{tag}. Additionally, this tuple has either I components $x_{I1}, x_{I2}, \ldots$, denoted x_I, or R components $x_{R1}, x_{R2}, \ldots$, denoted x_R. A true tag requests continued iteration with the I components bound to the iteration identifiers. A false tag requests return of the R components as the result of the iteration expression.

The iteration body evaluation function $\mathcal{E}_I$ of Iterbody $\rightarrow U \rightarrow V^*_\perp$ is defined like $\mathcal{E}$. In the definition of $\mathcal{E}_I$, in imitation of the environment updating notation, the tuple x with a true tag and I components x_I is denoted $\Lambda[\text{tag/true}][I/x_I]$. Similarly, the tuple x with a false tag and R components x_R is denoted $\Lambda[\text{tag/false}][R/x_R]$.

$\mathcal{E}_I[\![exp]\!]\rho = \mathcal{E}[\![exp]\!]\rho \Rightarrow \Lambda[\text{tag/false}][R/\mathcal{E}[\![exp]\!]\rho]$
$\mathcal{E}_I[\![\text{iter}(exp)]\!]\rho = \mathcal{E}[\![exp]\!]\rho \Rightarrow \Lambda[\text{tag/true}][I/\mathcal{E}[\![exp]\!]\rho]$
$\mathcal{E}_I[\![\text{let } idlist = exp \text{ in } iterbody \text{ end}]\!]\rho = \mathcal{E}[\![exp]\!]\rho \Rightarrow \mathcal{E}_I[\![iterbody]\!]\rho[idlist/\mathcal{E}[\![exp]\!]\rho]$
$\mathcal{E}_I[\![\text{if } exp \text{ then } iterbody_1 \text{ else } iterbody_2 \text{ end}]\!]\rho = \mathcal{E}[\![exp]\!]\rho \rightarrow \mathcal{E}[\![iterbody_1]\!]\rho, \mathcal{E}[\![iterbody_2]\!]\rho$

The least fixpoint operator Y is used to specify the iteration performed during evaluation of an iteration expression:

$$\mathcal{E}[\![\text{for } idlist = exp \text{ do } iterbody \text{ end}]\!]\rho = Y(\lambda F.\, \lambda x.\, x \Rightarrow (\mathcal{E}_I[\![iterbody]\!]\rho[idlist/x])_{tag} \rightarrow F((\mathcal{E}_I[\![iterbody]\!]\rho[idlist/x])_I), (\mathcal{E}_I[\![iterbody]\!]\rho[idlist/x])_R)\mathcal{E}[\![exp]\!]\rho$$

That is, each iteration is an evaluation of the iteration body with the iteration identifiers bound to some x and yields the tuple $\mathcal{E}_I[\![iterbody]\!]\rho[idlist/x]$. If the tuple has a true tag, its I components are bound to the iteration identifiers and iteration is resumed. If the tuple has a false tag, its R components are returned as the result of the iteration. The iteration identifiers are bound to $\mathcal{E}[\![exp]\!]\rho$ on the first iteration. If the iteration never terminates, evaluation is defined to "yield" $\perp$. The following easily proven lemma will be used in Section 4 to prove the consistency of the denotational and operational semantics of ADFL.

Lemma: If $\mathcal{E}[\![\text{for } idlist = exp \text{ do } iterbody \text{ end}]\!]\rho$ does not equal $\perp$, then there exists a sequence $\rho_1, \ldots, \rho_n$ such that for all i between 1 and n-1:

$\rho_1 = \rho[idlist/\mathcal{E}[\![exp]\!]\rho]$
$\rho_{i+1} = \rho_i[idlist/(\mathcal{E}_I[\![iterbody]\!]\rho_i)_I]$
$(\mathcal{E}_I[\![iterbody]\!]\rho_i)_{tag} = \text{true}$
$(\mathcal{E}_I[\![iterbody]\!]\rho_n)_{tag} = \text{false}$
$(\mathcal{E}_I[\![iterbody]\!]\rho_n)_R = \mathcal{E}[\![\text{for } idlist = exp \text{ do } iterbody \text{ end}]\!]\rho$

3. $\mathcal{O}\circ\mathcal{T}$: The Operational Semantics of ADFL

The operational semantics of an ADFL expression are a formal characterization of the behavior of the expression's data flow graph. The translation algorithm $\mathcal{T}$ is the "compiler" of ADFL. It maps expressions into their data flow graph implementations. The semantic function $\mathcal{O}$ maps graphs into functions representing their input-output behavior. $\mathcal{O}\circ\mathcal{T}$, the composition of these functions, is the operational semantics of ADFL.

In this section emphasis will be placed on the operational semantics of iteration expressions. In Section 4, the consistency of the denotational and operational semantics of ADFL will be proven using the productions of the BNF specification of ADFL as the inductive structure. Recall that, in the specification of the denotational semantics, the least fixpoint operator was used for only one production, that of the iteration expression. Likewise, in the specification of the translation algorithm, cyclic data flow graphs will be constructed for only one production, that of the iteration expression. Consequently, the iteration expression is the difficult and interesting case of the consistency proof, thus justifying our emphasis. Readers interested in a more detailed description of the operational semantics of ADFL may consult previous work of the author [5].

$\mathcal{T}$ maps expressions and $\mathcal{T}_I$ maps iteration bodies into their data flow graph implementations. The implementation of an 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. An expression graph has an output port, labeled by an integer, for each value returned by evaluation of *exp*. Recall the domain of the tuple $\mathcal{E}[\![iterbody]\!]\rho$. An iteration body graph has an output port tag for the tag; a set of I output port for results to be re-iterated; and a set of R output ports for results to be returned.

The semantic function $\mathcal{O}$ mapping data flow graphs into their operational characterization is defined using Kahn's [10] theory of parallel computation, which we briefly review.

Definition: The *history* of an operator or graph port is the sequence, possibly infinite, of values received or transmitted at that port during a data flow computation.

Theorem: Let V^ω denote the set of histories of data flow values. If V^ω is ordered so that $X \sqsubseteq Y$ if and only if X is a prefix of Y, then V^ω is a domain whose least element is the empty history ε.

Definition: The operational semantics of a data flow operator o are given by a continuous *history function* $\mathcal{O}[\![o]\!]$ 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 $\mathcal{O}[\![o]\!](X)$ represents the history of values produced at the output ports of o in response to X.

Note: Not all operators may be characterized by history functions. In particular, only determinate operators which for each input history tuple have only one possible output history tuple may be characterized thus. Since only determinate operators are used to construct graph implementations of ADFL expressions, the history function characterization is adequate for describing the operational semantics of ADFL.

The result of graph execution is defined to be the least fixpoint (solution) to a set of simultaneous equations, inferred from the history functions of the graph operators, whose variables represent the histories transmitted through the graph links.

In the remainder of this section, we will give a recursive definition of $\mathcal{O}\circ\mathcal{T}$ derived from fixpoint theory but will omit many details of the derivation. Also, the operational characterization of many ADFL expressions will be justified more by the actions of their data flow executions than by the structure of their data flow graph implementations. Readers desiring more detail knowledge of graph implementations should consult Brock [5].

Dennis [6], or Weng [13].

Note that the range and domain of $O\circ\mathcal{T}[\![exp]\!]$ differ from the range and domain of $\mathcal{S}[\![exp]\!]$. $O\circ\mathcal{T}[\![exp]\!]$ maps *history* environments, which are functions from identifiers to histories, into output history tuples. Viewed operationally, identifiers of ADFL are bound to histories of values.

The expression graphs, other than those for conditional and iterative expressions, have simple operational characterizations. Evaluation of an identifier yields the history to which the identifier is bound.

$$O\circ\mathcal{T}[\![id]\!]P = P[\![id]\!]$$

The data flow graphs $\mathcal{T}[\![const]\!]$ and $\mathcal{T}[\![oper(exp)]\!]$ contain the data flow operators *const* and *oper*. The operator *const* has a single input port, labeled trigger, and produces $\mathcal{I}[\![const]\!]$ whenever it receives the input value trigger. Whenever, the operator *oper* receives an input tuple x, it produces $\mathcal{I}[\![oper]\!](x)$.

$$O\circ\mathcal{T}[\![const]\!]P = O[\![const]\!](P[\![\text{trigger}]\!])$$
$$O\circ\mathcal{T}[\![oper(exp)]\!]P = O[\![oper]\!](O\circ\mathcal{T}[\![exp]\!]P)$$

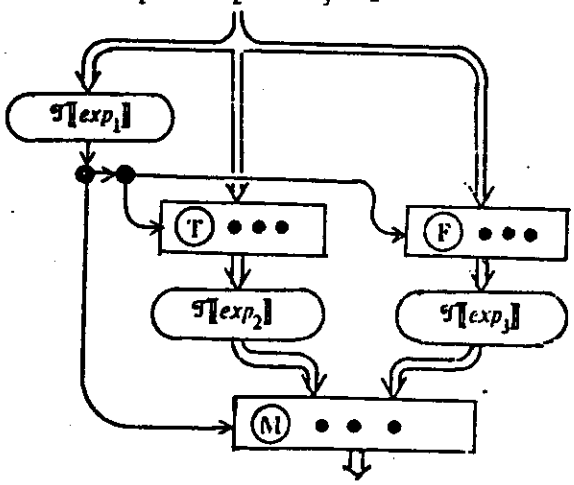
The denotational and operational semantics of ADFL differ in in their treatment of tuple expressions and binding expressions. The denotational semantics use the strict tuple concatenation operator $\|_s$ and strict identifier binding. The operational semantics use the usual tuple concatenation operator, $\|$, and non-strict identifier binding.

$$O\circ\mathcal{T}[\![exp_1, exp_2]\!]P = O\circ\mathcal{T}[\![exp_1]\!]P \,\|\, O\circ\mathcal{T}[\![exp_2]\!]P$$
$$O\circ\mathcal{T}[\![\text{let } idlist = exp_1 \text{ in } exp_2 \text{ end}]\!]P = O\circ\mathcal{T}[\![exp_2]\!]P[idlist/O\circ\mathcal{T}[\![exp_1]\!]P]$$

$\mathcal{T}[\![\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \text{ end}]\!]$, illustrated in Figure 2, contains a predicate subgraph $\mathcal{T}[\![exp_1]\!]$, a then expression subgraph $\mathcal{T}[\![exp_2]\!]$, a else expression subgraph $\mathcal{T}[\![exp_3]\!]$, and several *gates*. Each input value of the then expression must pass through a T gate and each input value of the else expression must pass through a F gate. The T gate has a control input port, a data input port, and a data output port. Each control value determines whether or not a data value may pass through the gate. If a true control value is received, a data value is absorbed and passed through the data output port. If a false control value is received, a data value is absorbed but not passed. In the F gate the sense of the control value is reversed. The output ports of the then expression subgraph and the else expression subgraph are paired by label, and each pair is joined by a M gate. The M gate has a control input port, two data input

---

Figure 2. $\mathcal{T}[\![\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \text{ end}]\!]$



ports, and a data output port. The control value determines which data value is passed through the gate. Connecting the control input ports of all these gates to the output of the predicate graph insures that the predicate can enable the execution of and select the results of the appropriate subexpression.

Let $\underline{T}$ be the composite history function of all the T gates used to implement the conditional expression. $\underline{T}$ maps a control history and a history environment into a history environment.

$\underline{T}(X, P)[\![id]\!] = O[\![T]\!](X, P[\![id]\!])$

With $\underline{F}$ and $\underline{M}$ denoting similar composite history functions, the operational semantics of the conditional expression may be expressed as:

$$O\circ\mathcal{T}[\![\textbf{if } exp_1 \textbf{ then } exp_2 \textbf{ else } exp_3 \textbf{ end}]\!]P = \underline{M}(O\circ\mathcal{T}[\![exp_1]\!]P,\ O\circ\mathcal{T}[\![exp_2]\!](\underline{T}(O\circ\mathcal{T}[\![exp_1]\!]P, P)),\ O\circ\mathcal{T}[\![exp_3]\!](\underline{F}(O\circ\mathcal{T}[\![exp_1]\!]P, P)))$$

The operational semantics of iteration bodies are a straightforward extension of the denotational semantics of iteration bodies and the operational semantics of expressions and will be discussed before the more onerous iteration expression is examined. Again, except for the conditional iteration body, the semantic equations are simple.

$O\circ\mathcal{T}_1[\![exp]\!]P = \Lambda[\text{tag}/O\circ\mathcal{T}[\![\text{false}]\!]P][R/O\circ\mathcal{T}[\![exp]\!]P]$

$O\circ\mathcal{T}_1[\![\text{iter}(exp)]\!]P = \Lambda[\text{tag}/O\circ\mathcal{T}[\![\text{true}]\!]P][I/O\circ\mathcal{T}[\![exp]\!]P]$

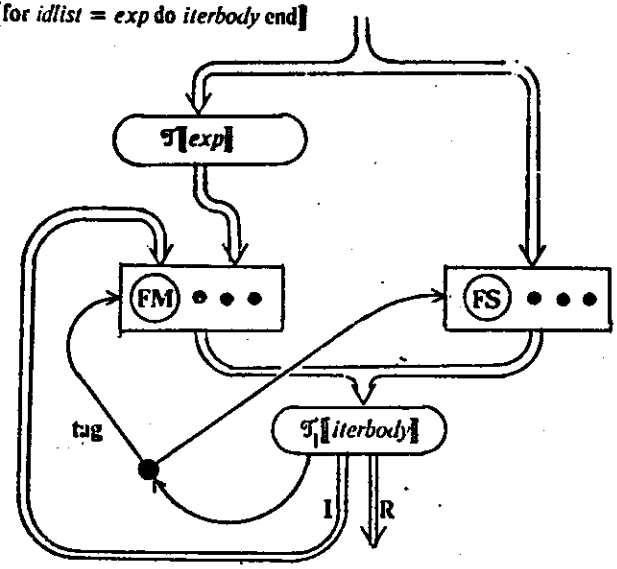
$O\circ\mathcal{T}_1[\![\textbf{let } idlist = exp \textbf{ in } iterbody \textbf{ end}]\!]P = O\circ\mathcal{T}_1[\![iterbody]\!]P[idlist/O\circ\mathcal{T}[\![exp]\!]P]$

The graph $\mathcal{T}[\![\textbf{if } exp \textbf{ then } iterbody_1 \textbf{ else } iterbody_2 \textbf{ end}]\!]$ resembles the graph, shown in Figure 2, of the conditional expression. However, because the conditional iteration body has three sets of output ports, a tag output port, I output ports and R output port; it has three sets of M gates, each set receiving a different control history. We leave to the most particular the wearisome task of defining the three appropriate control histories needed to specify $O\circ\mathcal{T}[\![\textbf{if } exp \textbf{ then } iterbody_1 \textbf{ else } iterbody_2 \textbf{ end}]\!]P$.

$\mathcal{T}[\![\textbf{for } idlist = exp \textbf{ do } iterbody \textbf{ end}]\!]$, illustrated in Figure 3, contains an initialization expression subgraph $\mathcal{T}[\![exp]\!]$, an iteration body subgraph $\mathcal{T}_1[\![iterbody]\!]$, FM gates, and FS gates. The FM gate is a M gate with an initial

---

Figure 3. $\mathcal{T}[\![\textbf{for } idlist = exp \textbf{ do } iterbody \textbf{ end}]\!]$



false control input. The FM gates select, under control of the tag values of the iteration body, either the outputs of the initialization expression or the R outputs of the iteration body. The selected values are sent to the iteration body input ports labeled by the iteration variables. The other iteration body inputs pass through FS gates. The FS gate absorbs, produces, and stores its data value whenever it receives a false control value. It produces its stored value, without absorbing a data value, whenever it receives a true control value. The FS gate has an initial false control value. Succeeding control values are the tag values of the iteration body. Thus, the FS gates store new values only when a "new" execution of the iteration body commences.

Let $\underline{\mathrm{FMS}}^{idlist}$ be the composite history function of the FM and FS gates. $\underline{\mathrm{FMS}}^{idlist}$ maps a quadruple consisting of the iteration body tag output, the iteration body I outputs, the initialization expression outputs, and the iteration expression history environment into the history environment input to the iteration body.

$$\underline{\mathrm{FMS}}^{idlist}(X, Y, Z, \mathrm{P})[\![id]\!] = \mathrm{O}[\![\mathrm{FM}]\!](X, Y_i, Z_i), \text{ if } id \text{ is the } i\text{-th element of } idlist$$
$$= \mathrm{O}[\![\mathrm{FS}]\!](X, \mathrm{P}[\![id]\!]), \text{ if } id \text{ is not an element of } idlist$$

The operational characterization of the iteration expression is obtained by deriving the least fixpoint to an equation constraining the outputs of the iteration body subgraph.

$$\mathrm{O}\circ\mathcal{T}_1[\![\textbf{for } idlist = exp \textbf{ do } iterbody \textbf{ end}]\!]\mathrm{P} = (\mathrm{Y}(\lambda X.\ \mathrm{O}\circ\mathcal{T}_1[\![iterbody]\!](\underline{\mathrm{FMS}}^{idlist}(X_{tag}, X_1, \mathrm{O}\circ\mathcal{T}[\![exp]\!]\mathrm{P}, \mathrm{P}))))_{\mathrm{R}}$$

The operational semantics of ADFL are certainly more complicated than the denotational. Furthermore, this complexity is not entirely the fault of our presentation, but rather largely the fault of the unusual conditions in which graphs containing non-terminating computations may produce results. Consider the following ADFL expressions with one free variable i:

for j = i **do if** j = 0 **then** iter(j) **else** j **end end**

which we abbreviate *DZ(i)*, for its evaluation diverges on zero values of i, and:

let k = *DZ(i)* **in if** i = 0 **then** 0 **else** k **end end**

which we abbreviate *IDZ(i)*, for its evaluation internally diverges on zero values of i. That is, when i is zero, although *DZ(i)* diverges, $\mathcal{T}[\![IDZ(i)]\!]$, the graph implementation *IDZ(i)*, "ignores" this internal divergence and yields zero. However, the ability to ignore divergence is limited. For example, if $\mathcal{T}[\![IDZ(i)]\!]$ receives the input sequence 0 • 1 as values of i, it will produce the output sequence 0. The second output cannot be produced until *DZ(i)* terminates its "computation" of the first output. Consequently, the ADFL expression:

for i = 0 **do**
 if *IDZ(i)* = 0 **then** iter(1) **else** i **end**
end

does not terminate, although it would if *IDZ(i)* freely ignored internal divergence.

The preceding example illustrates the intrinsic complexity of the operational semantics of ADFL and demonstrates the need for the simpler denotational semantics. In the next section, we will prove the consistency of the operational and denotational semantics.

4. The Consistency of ADFL

The operational and denotational characterizations of ADFL are consistent if they agree on all expression and iteration body evaluations defined to be non-terminating by the denotational semantics. We believe that it is quite reasonable to expect VAL programmers to only consider expressions which denotationally terminate to be correct. Expressions which terminate operationally, but not denotationally, waste resources in unnecessary computation.

Formally, the consistency requirement may be stated as:

$\mathcal{E}[\![exp]\!]\rho \neq \perp$ implies $\mathcal{O}\circ\mathcal{T}[\![exp]\!]\rho = \mathcal{E}[\![exp]\!]\rho$, and
$\mathcal{E}_I[\![iterbody]\!]\rho \neq \perp$ implies $\mathcal{O}\circ\mathcal{T}_I[\![iterbody]\!]\rho = \mathcal{E}_I[\![iterbody]\!]\rho$

To prove, by induction on the syntax of ADFL, the consistency requirement, a stronger consistency requirement is needed for the induction hypothesis, namely:

Given a sequence $\rho_1, \ldots, \rho_n$ of environments such that, for all i, $\mathcal{E}[\![exp]\!]\rho_i \neq \perp$:

$\mathcal{O}\circ\mathcal{T}[\![exp]\!]\rho_1 \bullet \ldots \bullet \rho_n = \mathcal{E}[\![exp]\!]\rho_1 \bullet \ldots \bullet \mathcal{E}[\![exp]\!]\rho_n$

Similarly, if, for all i, $\mathcal{E}_I[\![iterbody]\!]\rho \neq \perp$:

$\mathcal{O}\circ\mathcal{T}_I[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \rho_n = \mathcal{E}_I[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \mathcal{E}_I[\![iterbody]\!]\rho_n$

The proof of consistency is straightforward, but often tedious, for all BNF productions except the iteration expression, the only expression semantically characterized with the least fixpoint operator. For the simpler productions, the weaker consistency requirement easily implies the stronger. We will sample the inductive proofs of the simpler productions by proving the weaker consistency requirement for the binding expression.

Let ρ be an environment such that:

$\mathcal{E}[\![$**let** *idlist* = exp_1 **in** exp_2 **end**$]\!]\rho \neq \perp$.

The strictness of the denotational specification of the binding expression implies that $\mathcal{E}[\![exp_2]\!]\rho \neq \perp$. Consequently, using the weaker consistency requirement as the induction hypothesis, we know that:

$\mathcal{O}\circ\mathcal{T}[\![exp_2]\!]\rho = \mathcal{E}[\![exp_2]\!]\rho$

With successive applications of the definition of $\mathcal{O}\circ\mathcal{T}$, the weaker consistency requirement, the preceding equality, and the definition of $\mathcal{E}$, the desired case is proven.

$$\begin{aligned}\mathcal{O}\circ\mathcal{T}[\![\textbf{let } idlist = exp_1 \textbf{ in } exp_2 \textbf{ end}]\!]\rho &= \mathcal{O}\circ\mathcal{T}[\![exp_2]\!]\rho[idlist/\mathcal{C}\cdot\mathcal{T}[\![exp_1]\!]\rho] \\ &= \mathcal{E}[\![exp_2]\!]\rho[idlist/\mathcal{O}\circ\mathcal{T}[\![exp_1]\!]\rho] \\ &= \mathcal{E}[\![exp_2]\!]\rho[idlist/\mathcal{E}[\![exp_1]\!]\rho] \\ &= \mathcal{E}[\![\textbf{let } idlist = exp_1 \textbf{ in } exp_2 \textbf{ end}]\!]\rho\end{aligned}$$

Now we shall show how the stronger consistency requirement, used as an induction hypothesis, implies that the weaker consistency requirement holds for iteration bodies. Let ρ be an environment such that:

$\mathcal{E}[\![$**for** *idlist* = *exp* **do** *iterbody* **end**$]\!]\rho \neq \perp$

From, the lemma stated at the end of Section 2, we know that there exists a sequence $\rho_1, \ldots, \rho_n$ such that for all i between 1 and n-1:

$\rho_1 = \rho[idlist/\mathcal{E}[\![exp]\!]\rho]$
$\rho_{i+1} = \rho_i[idlist/(\mathcal{E}_1[\![iterbody]\!]\rho_i)_1]$
$(\mathcal{E}_1[\![iterbody]\!]\rho_i)_{tag} = \text{true}$
$(\mathcal{E}_1[\![iterbody]\!]\rho_n)_{tag} = \text{false}$
$(\mathcal{E}_1[\![iterbody]\!]\rho_n)_R = \mathcal{E}[\![\text{for } idlist = exp \text{ do } iterbody \text{ end}]\!]\rho$

Recall the operational characterization of the iteration body. Let F be the iteration evaluation function:

$$F = \lambda X.\, O\circ\mathcal{T}_1[\![iterbody]\!](\underline{\text{FMS}}^{idlist}(X_{tag}, X_1, O\circ\mathcal{T}[\![exp]\!]\rho, \rho))$$

Consequently:

$$O\circ\mathcal{T}[\![\text{for } idlist = exp \text{ do } iterbody \text{ end}]\!]\rho = Y(F)_R = (\sqcap F^i(\perp))_R$$

By induction on i, we may prove that:

$$F^i(\perp) = \mathcal{E}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \mathcal{E}_1[\![iterbody]\!]\rho_i = O\circ\mathcal{T}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \rho_i, \text{ if } i \leq n$$
$$F^i(\perp) = \mathcal{E}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \mathcal{E}_1[\![iterbody]\!]\rho_n = O\circ\mathcal{T}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \rho_n, \text{ if } i \geq n$$

For $i = 1$, we prove with successive applications of the definition of $F^1(\perp)$, the consistency requirement, the definition of $\underline{\text{FMS}}^{idlist}$ (recall its initial false control value), the definition of ρ_1, and the consistency requirement that:

$$\begin{aligned} F^1(\perp) &= O\circ\mathcal{T}_1[\![iterbody]\!](\underline{\text{FMS}}^{idlist}(\varepsilon, \varepsilon, O\circ\mathcal{T}[\![exp]\!]\rho, \rho)) \\ &= O\circ\mathcal{T}_1[\![iterbody]\!](\underline{\text{FMS}}^{idlist}(\varepsilon, \varepsilon, \mathcal{E}[\![exp]\!]\rho, \rho)) \\ &= O\circ\mathcal{T}_1[\![iterbody]\!]\rho[idlist/\mathcal{E}[\![exp]\!]\rho] \\ &= O\circ\mathcal{T}_1[\![iterbody]\!]\rho_1 \\ &= \mathcal{E}_1[\![iterbody]\!]\rho_1 \end{aligned}$$

For $i < n$, the induction hypothesis and the lemma of Section 2 imply that:

$$F^i(\perp)_{tag} = (\mathcal{E}_1[\![iterbody]\!]\rho_1)_{tag} \bullet \ldots \bullet (\mathcal{E}_1[\![iterbody]\!]\rho_i)_{tag} = \text{true}^i$$

By successive applications of the definition of ρ_i, the induction hypothesis, the definition of $\underline{\text{FMS}}^{idlist}$, and the preceding equality, we may conclude that:

$$\begin{aligned} \rho_1 \bullet \ldots \bullet \rho_{i+1} &= \rho \bullet \ldots \bullet \rho\,[idlist/\mathcal{E}[\![exp]\!]\rho \bullet (\mathcal{E}_1[\![iterbody]\!]\rho_1)_1 \bullet \ldots \bullet (\mathcal{E}_1[\![iterbody]\!]\rho_i)_1] \\ &= \rho \bullet \ldots \bullet \rho\,[idlist/\mathcal{E}[\![exp]\!]\rho \bullet F^i(\perp)_1] \\ &= \underline{\text{FMS}}^{idlist}(\text{true}^i, F^i(\perp)_1, \mathcal{E}[\![exp]\!]\rho, \rho) \\ &= \underline{\text{FMS}}^{idlist}(F^i(\perp)_{tag}, F^i(\perp)_1, O\circ\mathcal{T}[\![exp]\!]\rho, \rho) \end{aligned}$$

The definition of $F^{i+1}(\perp)$, the preceding equality, and the consistency requirement imply that:

$$\begin{aligned} F^{i+1}(\perp) &= O\circ\mathcal{T}_1[\![iterbody]\!](\underline{\text{FMS}}^{idlist}(F^i(\perp)_{tag}, F^i(\perp)_1, O\circ\mathcal{T}[\![exp]\!]\rho, \rho)) \\ &= O\circ\mathcal{T}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \rho_{i+1} \\ &= \mathcal{E}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \mathcal{E}_1[\![iterbody]\!]\rho_{i+1} \end{aligned}$$

Similarly, for $i \geq n$:

$$\begin{aligned} F^{i+1}(\perp) &= O\circ\mathcal{T}_1[\![iterbody]\!](\underline{\text{FMS}}^{idlist}(F^i(\perp)_{tag}, F^i(\perp)_1, O\circ\mathcal{T}[\![exp]\!]\rho, \rho)) \\ &= O\circ\mathcal{T}_1[\![iterbody]\!](\underline{\text{FMS}}^{idlist}(\text{true}^{n-1} \bullet \text{false}, F^i(\perp)_1, \mathcal{E}[\![exp]\!]\rho, \rho)) \\ &= O\circ\mathcal{T}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \rho_n \\ &= \mathcal{E}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \mathcal{E}_1[\![iterbody]\!]\rho_n \end{aligned}$$

Therefore, the above definitions of ρ_i and F imply:

$$\begin{aligned} O\circ\mathcal{T}[\![\text{for } idlist = exp \text{ do } iterbody \text{ end}]\!]P = Y(F)_R = (\sqcap F^i(\perp))_R &= (\mathcal{E}_1[\![iterbody]\!]\rho_1 \bullet \ldots \bullet \mathcal{E}_1[\![iterbody]\!]\rho_n)_R \\ &= \mathcal{E}[\![\text{for } idlist = exp \text{ in } iterbody \text{ end}]\!]\rho \end{aligned}$$

This proof can be extended to the stronger consistency requirement by observing that the control input $\text{true}^{n-1} \bullet \text{false}$ of $\underline{\text{FMS}}^{idlist}$ has reset $\underline{\text{FMS}}^{idlist}$ to its original state of waiting for inputs from outside the iteration expression. That extension completes the inductive proof of semantic consistency.