LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

An Operational Semantics for a Language with Early Completion Data Structures

CSG Memo 207

Jack B. Dennis

This research was supported by the National Science Foundation under research grant MCS-7915255.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

An Operational Semantics for a Language with Early Completion Data Structures[*]

Jack B. Dennis
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

In this paper we propose a formal operational approach to expressing the functional behavior of computer programs and systems that embody concurrency in their operation, and apply it to a functional programming language in which constructive operations on data structures have an "early completion" property. Our approach to formal operational semantics may be regarded as a simple alternative to the Vienna definition method, and incorporates a simple idea for representing the progress of interacting concurrent activities. It is suitable for modelling the semantics of programming languages or computer hardware at a fine grain level of parallelism. The language used in this study is APPL, an experimental, functional programming language under development at MIT as the base language for a general purpose data driven computer system.

We start by introducing the core of our textual language, which is a subset of the functional programming language VAL [1]. Next we present the basic idea of our formal operational semantics. We then introduce a class of binary trees as the data structures of our language and explain implementation of "early completion" of the tree constructor in terms of our operational model. We show how the extended language can be used to program classical stream-oriented computations in a natural way.

The present paper may be viewed as a continuation and refinement of earlier work reported in [2, 5].

The Core Language

For the core language we adopt a subset of the functional programming language VAL [1]. We include the basic data types null, boolean and integer, the basic expression forms

```
let <value - name> = <expression>
  in <expression>
endlet

if <expression>
  then <expression>
  {elseif <expression> then <expression> }
  else <expression>
endif
```

and the basic program form

```
function <function - name> ( <arguments> returns <result - types> )
  <expression>
endfun
```

We do not insist on type declarations except in function headings, since the type of any expression and, hence, the type of any defined value name, may be determined through flow analysis. Instead of providing an iteration construct, we permit functions to be invoked recursively. An occurrence of the function name in

[*]This research was supported by the National Science Foundation under research grant MCS-7915255.

body expression indicates a recursive invocation of the function. As an illustration of the language, we give a recursive program for the factorial of **n**:

```
funtion Fact ( n: integer returns integer )
  if n = 0 then 1
  else n * Fact ( n - 1 )
  endif
endfun
```

An Operational Semantics

For our operational semantics, we view a computation as progressing from configuration to configuration, where a configuration C consists of a state and a set of activities:

$$C = \langle S, A \rangle$$
$$A = \{a_1, .., a_n\}$$

where S is a State and each a_i is an Activity, and the two classes State and Activity are appropriate data types (sets). The relation between a configuration C = ⟨S, A⟩ and a successor configuration C' = ⟨S', A'⟩ is expressed by a function

Interp: State x Activity => State x **set**[Activity]

where

$$S' = \text{Interp}_1 (S, a)$$
$$A' = \text{Interp}_2 (S, a) \cup (A - \{a\})$$

and a is an arbitrarily chosen element of A.

The transition of a computation from one configuration to a successor configuration is interpreted as follows: The set A represents the concurrent activities present in the system being modelled. An arbitrary choice (a scheduling decision) of one of these, say a , is made as the activity whose progress is to be modelled by the current state transition. Applying the Interp function to this activity and the current state S yields a new state S' and a set of zero or more new activities that replace activity a in the new configuration C'.

An Interpreter for APPL

To present an interpreter for our textual language we suppose a program is represented as an abstract data flow graph expressed in VAL according to the following type definitions:

```
type Funtion = array [Instruction];

type Instruction = record [
  opc: Opcode;
  opd: array [Value];
  oct: integer;
  sct: integer;
  tgt: record [ normal, special: array [Target] ];
```

An instruction I consists of an operation code, an array I.opd containing any constant operands, a count I.oct of the number of operands required, a count I.sct of the number of signals required, and one or two arrays of targets which specify the inputs of instructions that are to receive the result value or a signal upon instruction completion.

```
type Opcode = oneof [
  Identity, Plus, Minus, Times: null;
  Equal, Greater, Less: null;
  Switch, Apply, Return: null ];
```

The additional opcodes for operations on binary trees will be given later.

```
type Value = oneof [
  nil: null;
  int: integer;
  boo: boolean;
  ret: record [
    act: Uid;
    tgt: array [Target] ] ];
```

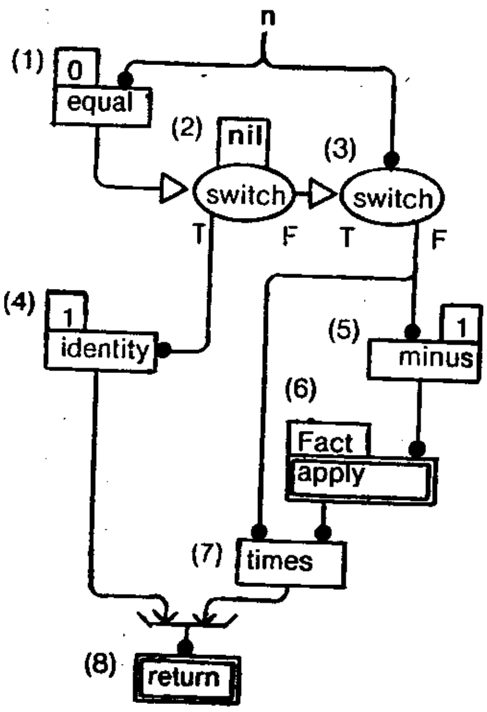
A value tagged "ret" represents a function return and specifies the instruction(s) to receive the result value or a signal. In a return value R, the unique identifier R.act specifies the activation from which the function was invoked.

```
type Target = record [
  typ: oneof [ val, sig: null ];
  ins: integer;
  inp: integer ];
```

In a target T the integers T.ins and T.inp specify, respectively, the index of the target instruction in the function array and the input port (operand number) of the instruction to which an operand value should be sent.

As an illustration of this encoding, the function array for the factorial example is shown in the form of a data flow graph [3] in Figure 1. Here, each box represents one instruction and is numbered by its index in the instruction array. The relation of the graph to the VAL format for instructions is straightforward. For example, instruction (7) is

```
record [
  opc: make Opcode [Times: nil ];
  opd: array - empty [Value];
  oct: 2;
  sct: 0;
  tgt: record [
    normal: [ 1: record [ins: 8; inp: 1] ];
    special: array - empty [Target] ] ]
```



The switch instruction (3) chooses between its two sets of targets according to the value of its boolean operand.

```
record [
  opc: make Opcode [Switch: nil ];
  opd: array - empty [Value];
  oct: 2;
  sct: 0;
  tgt: record [
    normal: array - empty [Target];
    special: [ 1: record [ins: 5; inp: 1];
             2: record [ins: 7; inp: 1] ] ] ]
```

The interpreter is designed so that each transition of configuration corresponds to execution of one instruction in the abstract program. So that many concurrent instances of function application may be in progress simultaneously, the values defined by each invocation must be represented in a data structure separate from the structure that represents the function itself. This data structure is called an Activation. The state of our interpreter for APPL models a heap in which the items are functions and activations:

```
type State = record [
  heap: array [Node];
  gen: integer ];

type Node = oneof [
  act: Activation;
  fcn: Funtion ];
```

The heap is an array of nodes indexed by a set of integers

```
type Uid = integer;
```

known as *unique identifiers*. The state component S.gen specifies the next integer available for use as a unique identifier.

```
type Activation = record [
  ins: array [Instance];
  fcn: Funtion ];

type Instance = record [
  opd: array [Value];
  oct: integer;
  sct: integer ];
```

An activation contains one instance component for each instruction activated during execution of the corresponding function. An instance I has an operand array component I.opd which accumulates operand values for the associated instruction. It also has count fields I.oct and I.sct which are decremented for each operand value or signal delivered to the instruction.

The set of activities in our interpreter models the set of instructions ready for execution:

```
type Activity = record [
  act: Uid;
  ins: integer ];
```

If A is an activity, then A.act is the unique identifier of the activation and A.ins is the index in the function of the instruction ready for execution. Note that our operational semantics models the concurrency of instructions within each function activation as well as the concurrency of instructions in different function activations.

The complete interpreter for APPL consists of the main function Interp and one semantic function for each opcode of the base language instruction set. The Interp function has the form

```
function Interp (
  A: Activity;
  S: State;
returns
  array [Activity], State )

  [ body ]

endfun
```

The body of Interp uses components of the activity record A to access the enabled instruction and its operands and selects the appropriate semantic function according to the opcode field of the instruction. Invocation of the semantic function builds the array of activities that replace A (these represent the instruction instances enabled by result packets and signals from execution of the chosen instruction.), and constructs the new state to reflect any creation or modification of items in the heap.

Binary Trees

To support program modules capable of producing output before having received all inputs, we present an operational semantics for binary trees in which the constructor operation **pair** transmits a representation of the result (a unique identifier) without waiting for the two components to arrive. We call these "early completion" data structures. This material has been influenced by the work of Kahn [6], Weng [7, 8], and Friedman and Wise [4].

To illustrate our realization of data structures, we extend the core language by adding binary trees as a data type. The representation of binary trees in the interpreter is as follows:

```
type Structure = record [
  l,r: Element ];

type Element = oneof [
  val: Value;
  que: Queue ];

type Queue = oneof [
  last: null;
  next: record [
    act: Uid;
    tgt: array [Target];
    que: Queue ] ];
```

If an element of a structure is a queue, it contains the targets of those instructions that attempt to read a tree component before it has been produced. A structure is a new kind of node in the interpreter's heap

```
type Node = oneof [
 ...
 id: Structure ];
```

and a tree value is represented by a unique identifier:

```
type Value = oneof [
 ...
 str: Uid ];
```

The opcodes for instructions operating on these representations are:

```
type Opcode = oneof [
 ...
 Pair, Mklft, Mkrht, Left, Right: null ];
```

A **pair** instruction creates a structure whose elements are empty queues. Instructions **left** and **right** select the corresponding component if it is a value, and append their activation identifier and targets to the queue otherwise. The **mklft** and **mkrht** instructions each replace a queue with a value and also transmit the same value to all target instruction instances specified in the queue.

Streams

The basic operations on streams of values are **cons**, **first**, **rest**, and the test **empty**. These operations have simple realizations using the representation

```
type Stream = oneof [
  empty: null;
  nonempty: record [ value: integer; tail: Stream ] ];
```

The realizations are given below where S is a stream and V is a value of type integer.

```
cons( V, S )
  The result is   make Stream [ nonempty: record [ value: V; tail S ] ].
first( S )
  The result is   S.value if S is not empty, and undefined if S is empty.
rest( S )
  The result is   S.tail if S is not empty, and undefined if S is empty.
empty( S )
  The result is   true if S = nil, and false otherwise.
```

The use of early completion for the construction of trees allows cascaded functions expressing a stream processing computation to execute concurrently. Classic examples such as the prime number sieve are easily expressed in the extended language.

Records

Records can be easily represented by binary trees: the translator maps field names into sequences in $\{ l, r \}^*$ in a consistent manner for each record type. Note that early completion of the record constructor operation follows from our implementation of binary trees. Therefore we will use multiple arguments and results in functions, assuming the implementation represents tuples of arguments and results by early completion records. Consequently, function evaluation commences with the arrival of any argument, and any result may be sent to its target instructions before all results of function application have been generated.

An Example

A classic programming example that illustrates the power of using streams in the construction of modular programs is the problem of testing whether the leaf elements of two trees determine the same sequence of values. This problem may be programmed in a nicely modular form by separately generating the "fringe" of each tree in the form of a stream of values. The streams are compared element-by-element until a mismatch is found or one of the streams ends, and the result is **false**; if the comparison consumes both streams simultaneously the result is **true**. The point is that without concurrent production and consumption of streams, the entire fringe of each tree would be generated before the comparison is started.

Our solution in APPL involves a function Fringe that generates the fringe of a tree, a function Test that compares two fringes(streams) for equality, and functions that implement the operations of building a single-element stream and concatenating two streams. Definitions of these functions in APPL are given below.

```
function EqualFringe ( A, B: Tree returns boolean )
  type Tree = oneof [
    atom: integer;
    tree: record [ lft,rht: Tree ] ];
  type Stream = oneof [
    empty: null;
    nonempty: record [
      value: integer;
      tail: Stream ] ]

  function Fringe ( T: Tree returns Stream )
    tagcase T
      tag tree: Catenate ( Fringe (T.lft), Fringe (T.rht) )
      tag atom: MakeStream (T)
    endtag
  endfun
```

```
    function MakeStream ( n: integer returns Stream )
      make Stream [ nonempty:
                 record[ value: n;
                        tail: make Stream[ empty: nil ] ]
    endfun

    function Test ( X, Y: Stream returns boolean )
      tagcase X
        tag empty:
          tagcase Y
            tag empty: true
            tag nonempty: false
          endtag
        tag nonempty:
          tagcase Y
            tag empty: false
            tag nonempty:
              ( X.value = Y.value ) & Test ( X.tail, Y.tail )
          endtag
      endtag
    endfun

    function Catenate ( X, Y: Stream returns Stream )
      tagcase X
        tag empty: Y
        tag nonempty:
          make Stream [ nonempty: record [
            value: X.value;
            tail: Catenate ( X.tail, Y ) ] ]
      endtag
    endfun

    Test ( Fringe (A), Fringe (B) )
  endfun
```

The coding of the two key functions, MakeStream and Catenate, in the representation of the interpreter is shown in Figure 2. In these routines streams are represented as binary trees where the left subtree is the first element of the stream, the right subtree is the rest of the stream, and the end of a stream is denoted by the special value **nil**. Note that in the code for Catenate, a partial structure is returned immediately, without waiting for any recursive invocations of Catenate to terminate.

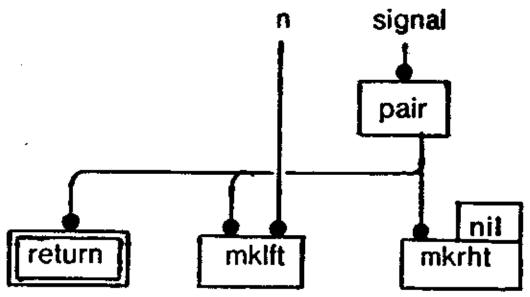


Figure #2a   Data flow graph for MakeStream

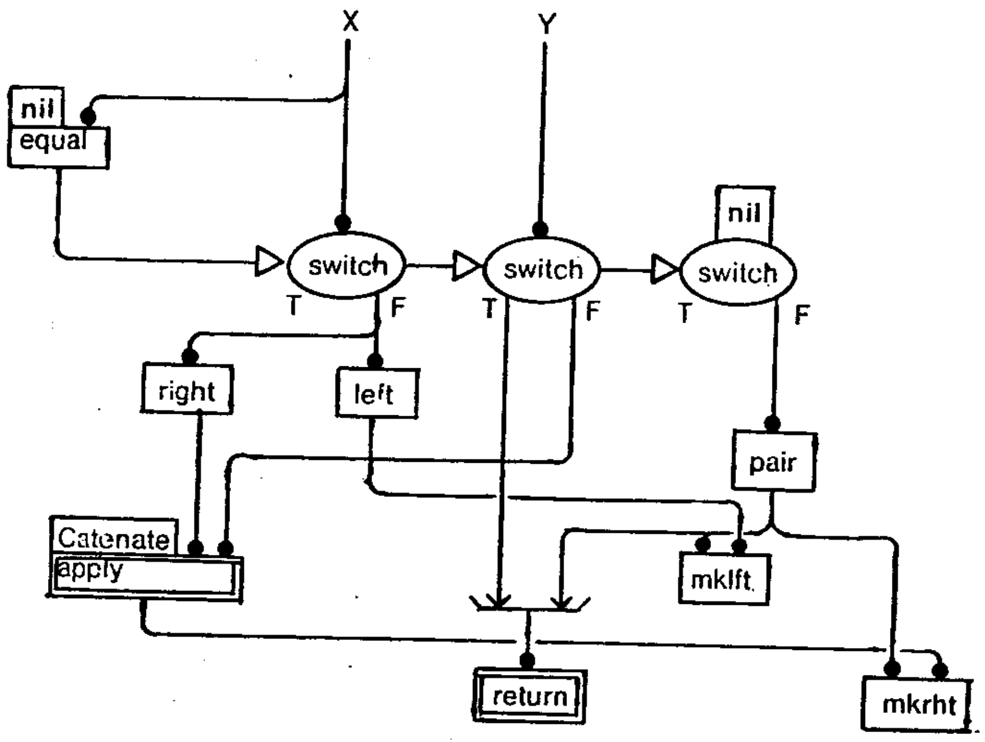


Figure # 2b Data flow graph for Catenate

Remarks

The formal interpreter presented in this paper is intended as the semantic model for an experimental computer system. The language of this paper, extended to include support for expressing nondeterminate computations, will be the base language of the envisioned computer. The implementation will exploit the concurrency of activities in the formal interpreter to achieve a high level of concurrency of data transfers within a hierarchical memory system.

References

1. Ackerman, W. B. and Dennis, J. B. VAL-A Value Oriented Language. Tech.Rep. TR-218, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, December, 1978.

2. Arvind, K.P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Tech.Rep. TR-114a, Department of Information and Computer Science, University of California-Irvine, Irvine, California, December, 1978.

3. Dennis, J.B. First Version of a Data Flow Procedure Language. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, B. Rodiner, Ed., Springer-Verlag, *Lecture Notes in Computer Science*, Vol 19, 1974, pp. 362-376.

4. Friedman, D.P., and Wise, D.S. CONS Should Not Evaluate its Arguments. In *Automata, Languages, and Programming*, July 1976, pp. 257-284.

5. Henderson, D.A. The Binding Model: A Semantic Base for Modular Programming Semantics. Tech.Rep. TR-145, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, February, 1975.

6. Kahn, G., and D. MacQueen. Coroutines and Networks of Parallel Processes. Information Processing 77: Proceedings of IFIP Congress 77, August 1977, pp. 993-998.

7. Weng, K.-S., Stream-Oriented Computation in Recursive Data Flow Schemas. Tech.Rep. TM-68, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, October, 1975

8. Weng, K.-S., An Abstract Implementation for a Generalized Data Flow Language. Tech.Rep. TR-228, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, 1979