VIM: An Experimental Multiuser System Supporting Functional Programming

Computation Structures Group Memo 238
Massachusetts Institute of Technology
Cambridge, MA 02139
9 May 1984

Jack B. Dennis, Joseph E. Stoy
and
Bhaskar Guharoy

To be presented at the International Workshop On High-Level Computer Architecture at Los Angeles, California, May 23-25, 1984.

VIM: An Experimental Multi-User System Supporting Functional Programming

by
Jack B. Dennis, Joseph Stoy
and
Bhaskar Guharoy

Abstract: *An experimental, multi-user computing system to support functional programming is being developed using principles of data flow computer architecture. Users of* VIM *will write programs in* VIMVAL, *an extension of the strongly-typed language* VAL *to support streams, recursion, and high-order functions. Program modules are compiled into function templates which encode data flow graphs as groups of* VIM *machine instructions. The design of* VIM *provides special support for streams using early-completion data structures and suspensions, and tail recursion is used to avoid unnecessary retention of activation records. This paper discusses the experimental system in general and explains the program execution mechanisms of* VIM *in terms of an example program module.*

The ideas discussed in this paper were developed with support from the National Science Foundation, the U. S. Department of Energy, and the Advanced Research Projects Agency of the U. S. Department of Defense.

1. Introduction

This paper reports on a project in the Computation Structures Group of the MIT Laboratory for Computer Science to develop an experimental computer system embodying concepts of functional programming and data flow computer architecture. The VIM (VAL *Interpretive Machine*) is a prototype for a multi-user computer system that supports the functional programming style [5] for a user community sharing a collection of information and program modules. This project is the fruition of a research program that has evolved from concepts of computer system organization presented in [8]. Related efforts toward practical computer systems supporting functional programming include [4, 17, 23, 6, 3, 16]. The goals for the VIM Project are discussed in [10] and the semantic model on which the implementation is based is sketched in [11].

We intend that programs to be run on VIM will be expressed, directly or indirectly, in the high-level functional programming language VIMVAL. We believe VIMVAL is sufficiently complete to provide access to all essential features of the computer system. Hence it should not be necessary for users to depart from expressing information processing applications in VIMVAL for any requirement of their application, be it a text editor or an airline reservation system. Thus VIM does not provide any facilities other than those of the VIMVAL language for the manipulation of data bases or for the control of concurrent tasks.

Section 2 of the paper illustrates the VIMVAL language through a classical example of programming with streams – testing two trees for equality of their "fringes". In Section 3, this example is used to illustrate the program execution mechanisms of VIM – tail recursion to avoid unnecessary retention of activation records, records with early completion to implement streams, and suspensions to implement demand-driven generation of stream elements. Section 4 discusses modules, binding and type checking, and Section 5 reviews progress and plans of the project.

2. The VIMVAL Language

The programming language for the VIM system is VIMVAL, an applicative language which is a revision and extension of the VAL programming language [1, 2, 20]. The extensions include the addition of stream-types, free variables, recursion and mutual recursion, and treating functions as first-class objects. The requirements for type declaration have been relaxed as discussed in Section 4.

An example of a program module written in VIMVAL is shown in Figure 1. It consists of a header specifying its interface, several function definitions, and an expression that constitutes the function body. This module is chosen to illustrate how certain features of VIMVAL are supported by the execution mechanisms of VIM discussed in Section 3. It returns a record whose fields contain functions that may be used to build trees and to test if their "fringes" are equal. The definitions of the two functions of particular interest are presented in Figures 2 and 3. In the following paragraphs we explain the main features of VIMVAL.

A module written in VIMVAL defines a function that may be invoked from within another module or by a user command to the system (see Section 4). A module may contain internal function definitions – these may be invoked only from within the module unless they are

2

```
module returns operations;
    type operations = record[build, equal : function];
    type CharStream = stream[char];
    type TreeType = oneof [node: record[left, right : TreeType];
                           leaf : char];

    function EqualFringe(t1, t2 : TreeType returns boolean)
        function EqualStream (s1, s2 : Charstream returns boolean)
            % The body of function EqualStream
            % is shown in Figure 2.
        endfun

        EqualStream(Leaves(s1), Leaves(s2))
    endfun;

    function Leaves(t: TreeType returns CharStream)
        % The body of function Leaves
        % is shown in Figure 3.
    endfun;

    function CreateTree (s : CharStream returns TreeType)
        .
        .
    endfun;

    % The body expression of the module follows.

    record [   build : CreateTree,
               equal : EqualFringe]

endmodule;
```

Figure 1: Text of a program module written in VIMVAL. The bodies of the function definitions are omitted for simplicity.

incorporated into data structures sent out as module results. The body of a module may use names that are not defined bound to values by definitions in the module. These *free* names must be bound to other modules before the module may be run.

The data types of VIMVAL fall into two classes – *simple* types and *structure* types. The simple types include the familiar types **integer**, **real**, **boolean**, **character** and **null**. The structure types include array-types, record-types, distinguished unions, stream-types, and functions. The definition of a record-type takes the form

type *Node* = **record** [*left*, *right* : *Tree*];

Records of type *Node* have two fields named *left* and *right*. A record may be constructed by the record building operation

record [*left*: *t1*, *right*: *t2*]

where *t1* and *t2* are of type *TreeType*. Record fields are accessed by record selection, for instance

N.left

where *N* is of type *Node*. A distinguished union type is used where different choices of representation are appropriate for different cases of a value. The trees used in our example conform to the type definition

```
type TreeType = oneof [node: record[left, right: TreeType];
                                                leaf: char]
```

where the two subtypes are distinguished by the tags *node* and *leaf*. A case expression is used to access values of a **oneof** type:

```
tagcase N
tag node:  expr1;
tag leaf:  expr2;
endtag
```

The tag of value *N* determines which of *expr1* and *expr2* are to be evaluated.

A stream is a sequence of values, all of the same type. A stream may be unending, as in the stream of characters received from a terminal keyboard. The definition

type *CharStream* = **stream [character]**

defines the type *CharStream* to be stream of characters. The operation

empty[*Charstream*]

creates an empty stream. The other operations defined for streams are **first**, **rest**, and **affix**. If *S* is of type **stream** [*T*] and *v* is a value of type *T* then **first**(*S*) gives the value of the first element of the stream, **rest**(*S*) returns the stream *S* without its first element, and **affix**(*v*, *S*) returns the stream *S* prefixed by *v*.

Functions are first-class objects. They may be passed as arguments to and returned as results from functions, and they may be built into data structures. The body of a function definition is an expression. Evaluation of an expression yields a single value or a tuple of values. Forms of an expression include the conditional expression, the **tagcase** expression illustrated above, and function invocation. The group of functions defined in a module may be recursive or mutually recursive. There is no form of expression for writing conventional iteration, use of recursion being preferred.

Figure 2 shows the definition of the function *EqualStream* that tests equality of two streams. Note that once an unequal pair of characters is tested, the remaining elements of the argument streams are irrelevant to determining the result. The use of demand-driven computation of stream elements in VIM avoids computation of these unneeded elements. The function *Leaves* in Figure 3 converts a tree into the stream of characters found in a left-to-right traversal of the leaves of the tree. It uses the function *StreamOfLeaves* which descends the leftmost path in the tree *t* appending to the list *c* the right-branching subtrees it encounters on the way. After reaching the leaf at the end of a path, the most recently added subtree in the continuation list is processed in the same manner.

```
function EqualStream(s1, s2: CharStream returns boolean)

      if null(s1) and null(s2) then true
      elseif null(s1) or null(s2) then false
      elseif first(s1) = first(s2)
          then EqualStream(rest(s1), rest(s2))
      else false
      endif
endfun;
```

Figure 2: Definition of the function *EqualStream*.

```
function Leaves ( t: TreeType returns CharStream )

type Continuation = oneof [
      end: null,
      element: record [next: TreeType, rest: Continuation]]

   FunctionStreamOfLeaves( t: TreeType, c: Continuation)
                  returnsCharStream)

      tagcase t
      tag node:
            StreamOfLeaves (t.left,
                  make Continuation [element:
                        record [ next: t.right, rest: c]])
      tag leaf:
            tagcase c
            tag end: affix (t, empty )
            tag element:
                  affix(t, Leaves ( c.next, c.rest ) )
            endtag
      endtag
      endfun

      StreamOfLeaves(t, make Continuation [end: nil])
endfun
```

Figure 3: Definition of the function *Leaves*.

Some beautiful examples of the use of streams and demand-driven computation have been given by Turner [24].

5

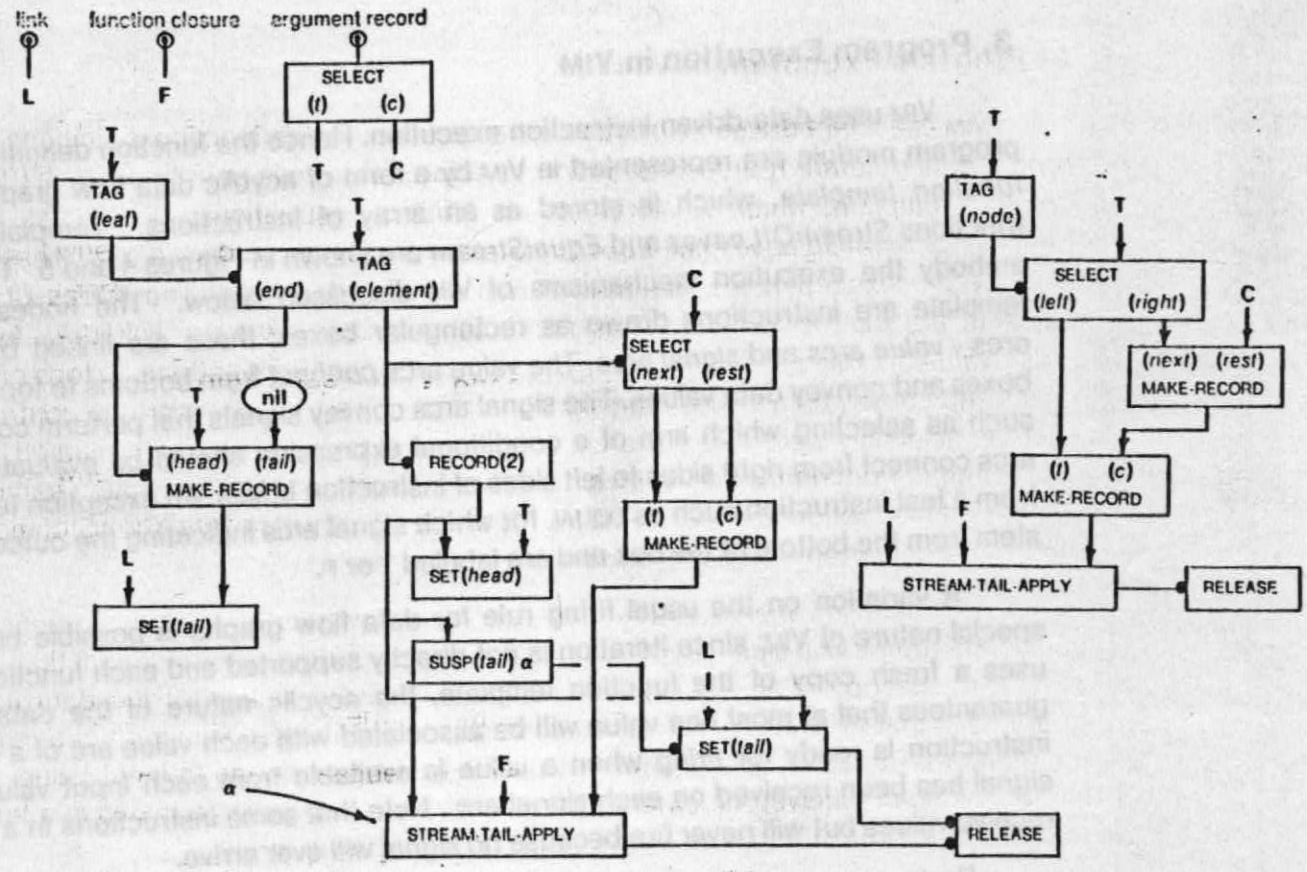

Figure 4: Function template for *StreamOfLeaves*. If the given tree is a leaf and the continuation is not empty (tag = *element*), the stream is extended by adding a record in which the *tail*-field is made a suspension (by the SUSP instruction). The suspension contains an address *α* of the STREAM-TAIL-APPLY instruction which will be activated by the consumer of the stream.

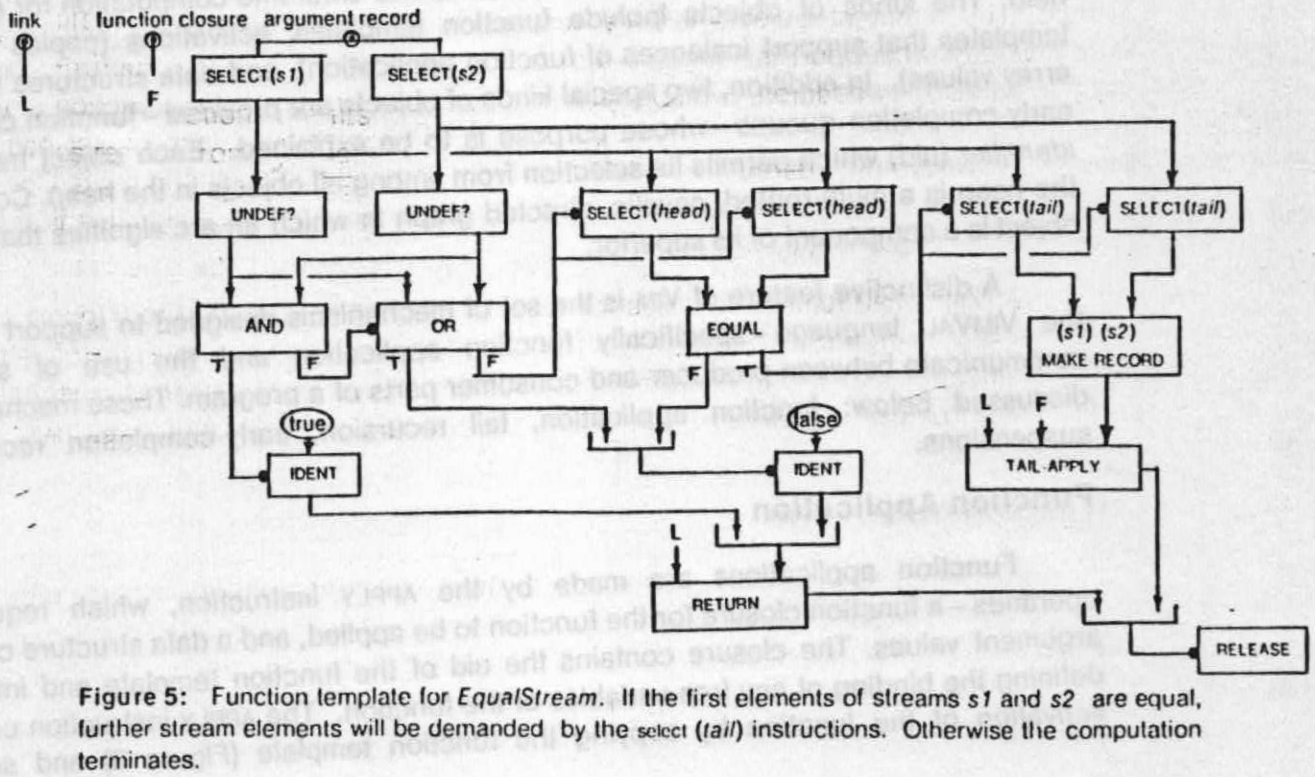

Figure 5: Function template for *EqualStream*. If the first elements of streams *s1* and *s2* are equal, further stream elements will be demanded by the select (*tail*) instructions. Otherwise the computation terminates.

3. Program Execution in VIM

VIM uses data-driven instruction execution. Hence the function definitions of a VIMVAL program module are represented in VIM by a form of acyclic data flow graph [9, 7] called a *function template*, which is stored as an array of instructions. Templates for the two functions *StreamOfLeaves* and *EqualStream* are shown in Figures 4 and 5. These templates embody the execution mechanisms of VIM discussed below. The nodes of a function template are instructions drawn as rectangular boxes; these are linked by two kinds of arcs - *value arcs* and *signal arcs*. The value arcs connect from bottoms to tops of instruction boxes and convey data values. The signal arcs convey signals that perform control functions such as selecting which arm of a conditional expression should be evaluated. The signal arcs connect from right sides to left sides of instruction boxes. An exception to this is signals from a test instruction such as EQUAL for which signal arcs indicating the outcome of the test stem from the bottom of the box and are labeled T or F.

A variation on the usual firing rule for data flow graphs is possible because of the special nature of VIM: since iteration is not directly supported and each function application uses a fresh copy of the function template, the acyclic nature of the data flow graphs guarantees that at most one value will be associated with each value arc of a template. An instruction is ready for *firing* when a value is available from each input value arc, and a signal has been received on each signal arc. Note that some instructions in a template will receive values but will never fire because no signal will ever arrive.

During operation of VIM, many function applications will be active simultaneously, and the machine is free to choose instructions for execution from any active template so long as the firing rule is observed.

VIM maintains a *heap* in which all objects that enter into computation for any user are held. The kinds of objects include function templates, activations (copies of function templates that support instances of function application), and data structures (record and array values). In addition, two special kinds of objects are provided - function closures and early-completion queues - whose purpose is to be explained. Each object has a *unique identifier* (uid) which permits its selection from among all objects in the heap. Conceptually, the heap is a multi-rooted, acyclic, directed graph in which an arc signifies that the target object is a component of its superior.

A distinctive feature of VIM is the set of mechanisms designed to support aspects of the VIMVAL language - specifically function application and the use of streams to communicate between producer and consumer parts of a program. These mechanisms are discussed below: function application, tail recursion, early-completion records, and suspensions.

Function Application

Function applications are made by the APPLY instruction, which requires two operands - a function closure for the function to be applied, and a data structure containing argument values. The closure contains the uid of the function template and information defining the binding of any free variables of the function. The APPLY instruction creates an *activation* of the function by copying the function template (Figure 6) and sends the argument structure and a *return-link* to the template copy. The return-link is the address of the target instruction, the instruction which is to receive the result of function application. It consists of the uid of the calling activation and the index of the target instruction in the function template.

Instructions of the activation are then executed according to the data flow firing rule until the RETURN instruction is enabled. The effect of the RETURN instruction is to send the result value to the instruction specified by the return link. A separate RELEASE instruction returns the storage occupied by the function template to the free storage pool of VIM. This function is separate from the RETURN instruction because execution of the RETURN instruction is, as we shall see, not always the last event of an activation.

Each activation requires some memory for its representation in the computer. The amount of memory required by a program varies dynamically as activations are created and discarded. The RELEASE instruction is the last instruction to be executed in an activation and it releases the memory occupied by the activation so that it may be reused.

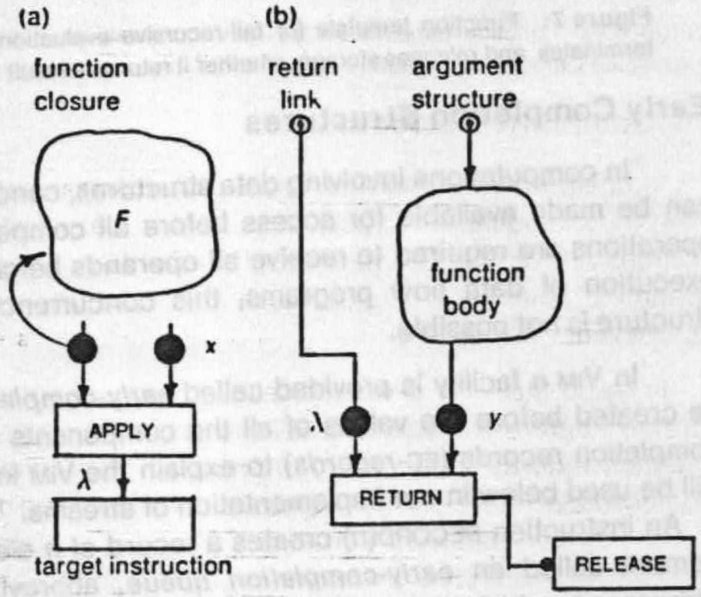

Figure 6: The APPLY instruction and function invocation. (a) The APPLY instruction, ready to fire, and its target instruction. (b) Activation of *F*, ready to send result value *y* to the target instruction and release storage.

Tail Recursion

In many cases the value returned by a function *f* is computed directly by a recursive application of *f*, as shown in Figure 7. In this situation the result to be returned by the caller is exactly that returned by the callee, and reactivation of the caller is unnecessary. VIM has a special instruction TAIL-APPLY that implements this. It acts like APPLY but has an extra operand, a return-link which it passes to the function template instead of generating a new one.

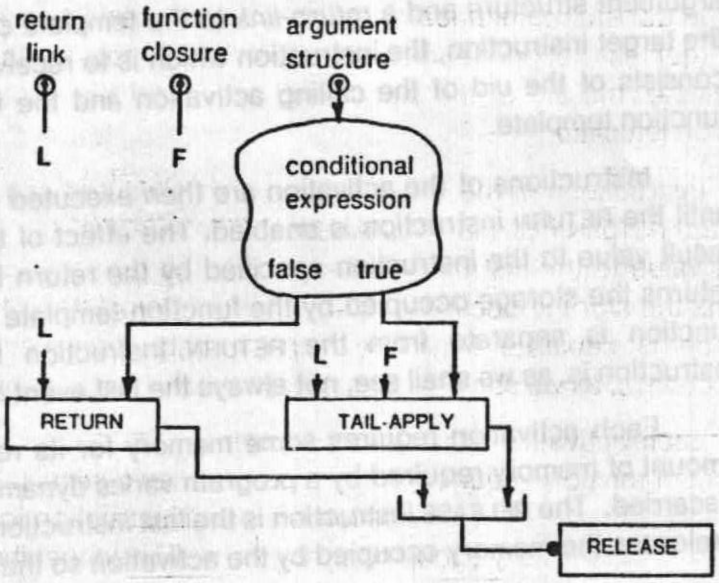

Figure 7: Function template for tail-recursive evaluation of a function. Note that each activation terminates and releases storage, whether it returns a result or invokes itself recursively.

Early Completion Structures

In computations involving data structures, concurrency is increased if a data structure can be made available for access before all component values have been computed. If operations are required to receive all operands before their application, as is usual for the execution of data flow programs, this concurrency of creating and accessing a data structure is not possible.

In VIM a facility is provided called *early-completion structures* to permit structures to be created before the values of all the components are available. Here we will use early-completion records (EC-*records*) to explain the VIM implementation because these records will be used below in our implementation of streams. The mechanism is illustrated in Figure 8. An instruction RECORD(n) creates a record of *n* elements, each initialized with a special element called an *early-completion queue*, abbreviated ECQ. An ECQ holds a set of addresses to which the value of the record field must be sent once available. Whenever the field is filled in by a SET instruction, the ECQ is replaced with its value which is also sent to all instructions with addresses in the queue. If a SELECT instruction attempts to access the field while it is an ECQ, the address of the SELECT instruction is entered in the queue.

The early-completion mechanism makes it possible to allow function applications to begin execution before the values of all their arguments have been computed. This is done by packaging the function's arguments into an EC-record. Similarly, the result values, if more than one, may be returned as an EC-record so each may be available to the caller without waiting for all results to be evaluated.

The Implementation of Streams

An attraction of using streams is that the producer and consumer of a stream can operate concurrently. In our example of testing the fringes of two trees, the consumer *EqualFringe* may begin processing pairs of stream elements as soon as the first pair has

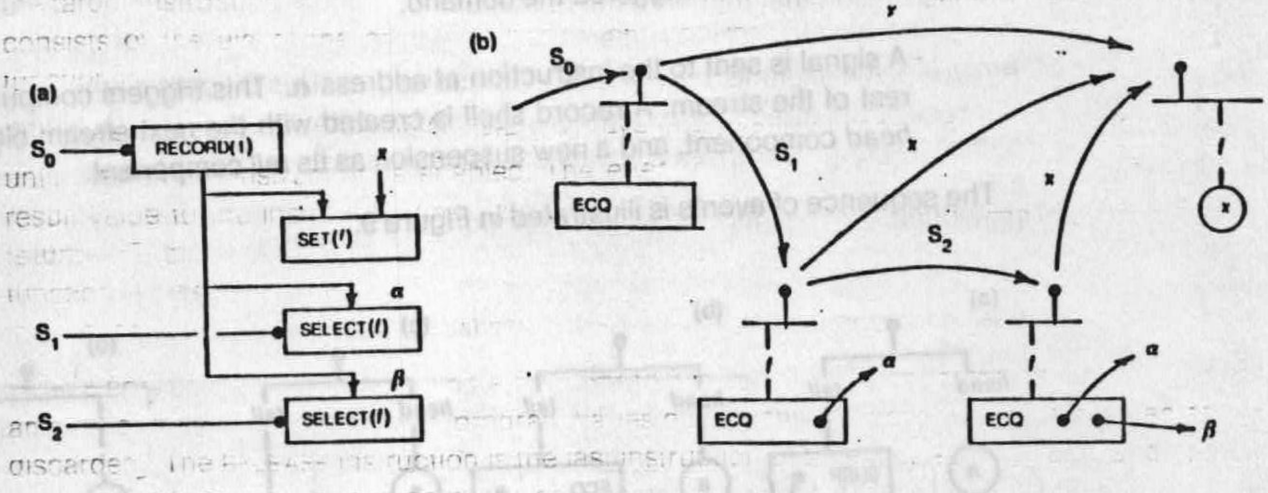

Figure 8: Use of the early-completion mechanism for a record field. (a) Typical coding. (b) Transitions of an EC-record. The RECORD and SELECT instructions are activated by signals s_0, s_1, and s_2; x is the value that activates the SET instruction; and α and β are the addresses of the SELECT instructions.

been produced by two activations of *Leaves*. Meanwhile the producers may continue execution to generate further stream elements. To achieve this effect, a stream in VIM is represented by a chain of EC-records:

stream[*T*] = **record**[*head*: *T*, *tail*: **stream**[*T*]]

The *head* field holds a stream element and the *tail* field holds the remainder. A function that produces a stream passes to the consumer function an EC-record with an ECQ as its *tail* component and continues to generate the next record in the chain which it puts in place of the ECQ. The consumer proceeds down the chain of records, waiting whenever it encounters an ECQ until a value is supplied.

This data-driven scheme permits the producer to get ahead of the consumer by an unbounded distance, using storage for the portion of the stream that has been produced but not consumed. Even worse, in the case of *EqualFringe*, once an unequal pair of stream elements is encountered, the remainders of the streams are irrelevant, but nothing stops their production.

In VIM such wasteful computation is avoided by processing streams in a *demand-driven* manner: An element of the stream is computed only if the consumer demands its value. Demand-driven evaluation is also known as lazy evaluation [12] or delayed evaluation.

Demand-driven evaluation of streams in VIM is implemented using a special record element called a *suspension*. A suspension contains the address α of the instruction of the stream producer which would trigger computation of the next element of the stream. When the consumer tries to access the next element using a SELECT instruction at address β, it will find a suspension. Then VIM performs two actions:

- The suspension is replaced by an ECQ containing the address β of the instruction that triggered the demand.

- A signal is sent to the instruction at address α. This triggers computation of the rest of the stream. A record shell is created with the next stream element as its *head* component, and a new suspension as its *tail* component.

The sequence of events is illustrated in Figure 9.

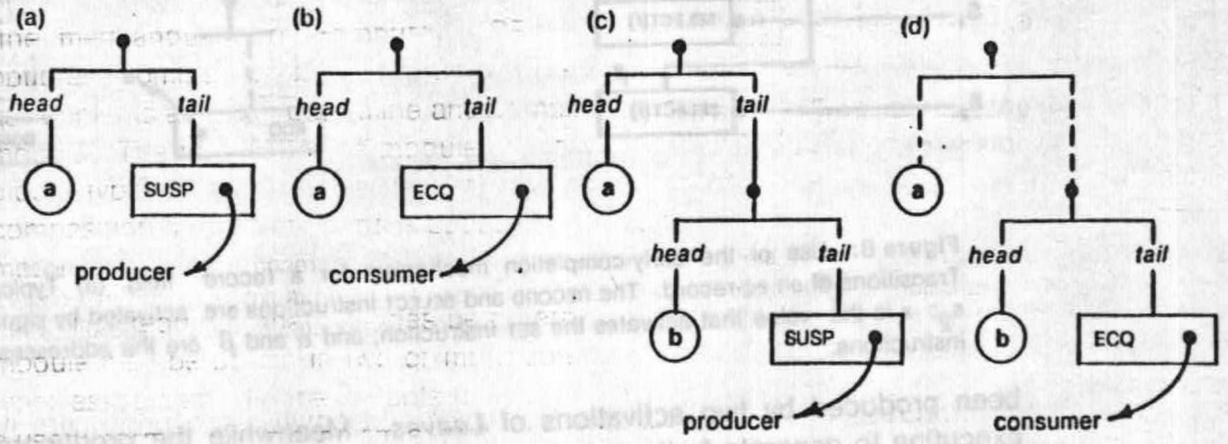

Figure 9: Demand-driven generation of stream elements. (a) Stream element; the producer is awaiting a demand. (b) The consumer demands the next stream element. (c) The producer generates one stream element and suspends itself. (d) The consumer abandons the previous element and demands another.

Figure 10 shows how tail recursion can be used to advantage in functions that produce streams. Typically, a stream producer defines its result using an **affix** operator as in the following code outline:

```
function F( a: T returns stream[T] )
    let v = H(a); x = G(a);
    in affix(v, F(x) )
    endlet
endfun;
```

In this outline *v*, the next stream element, is defined by function *H*, and the remaining stream elements are computed by a recursive application of *F*. By usual convention, this is not tail recursion because the **affix** operator is applied to the result returned by *F*. Nevertheless the advantage of tail recursion can be reaped by using an EC-record as the result returned by *F* and letting the recursive activation of *F* perform the **affix** operation by filling in the *tail* component. The coding for this scheme is shown in Figure 10. Instructions in function template F1 fill in the EC-record passed to it as a *data-link*, and from a new EC-record to pass on to a recursive activation of itself. The function template for F simply creates an EC-record which it both returns and passes as the data-link of the initial activation of F1.

The implementation shown for *StreamOfLeaves* in Figure 4 is a slight elaboration of

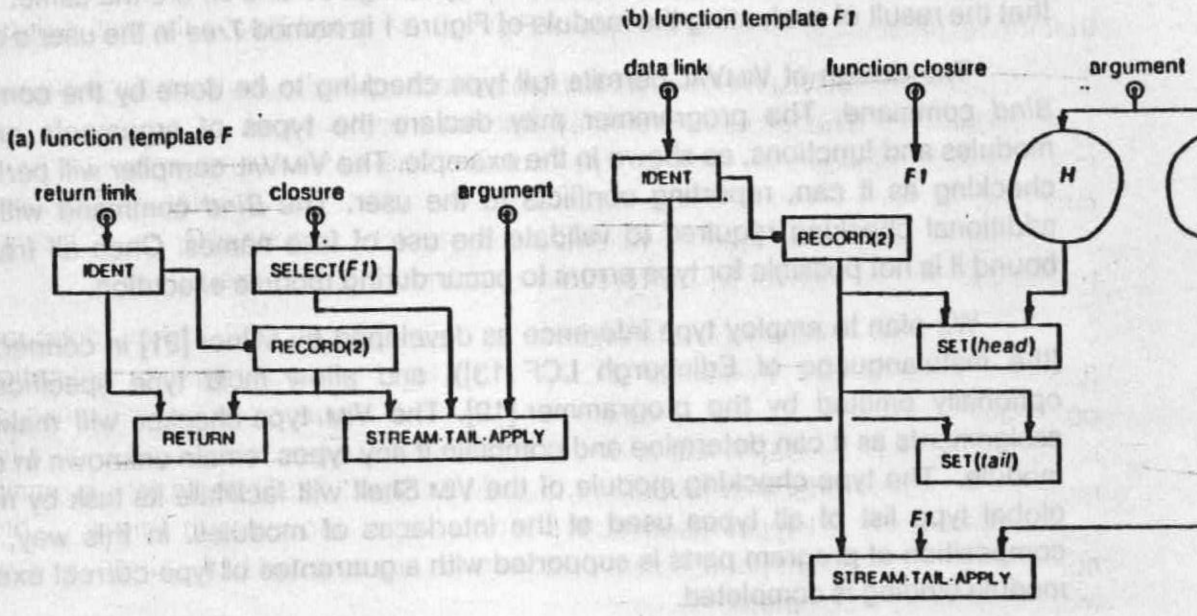

Figure 10: Function templates for tail-recursive implementation of a stream producer.

this code to implement demand-driven operation using suspensions.

4. Modules and the User Shell

Users will communicate with VIM by giving commands to be executed by a group of modules collectively called the *Shell*. We anticipate that these modules will be programmed in VIMVAL.

The functions of the Shell include maintaining a directory of named data structures and program modules for each user, and providing the interface through which users create program modules, compile them, bind them into executable programs, and request the performance of computation.

To compile a program module the user will type

Define (*M*, *Translate* (*P*))

This defines the value of *M* to be the result of invoking the VIMVAL compiler to translate *P*.

The free names of a module must be bound to other modules before the module may be run. If *f* is a free name of *P*, the user may define a version *P1* in which *f* is bound to module *F* by giving the command

Define (*P1*, *Bind* (*P*, *f*, *F*))

If *R* has no free names, it may be executed by typing

Define (*y* = *R* (*x1*, ... , *xm*))

where the arguments *x1*, ... , *xm* are literal values or names of objects, and the result will become the value of directory name *y*. More specifically, the user might type

```
let t1 = Tree.build ( s1 );
    t2 = Tree.build ( s2 );
in  Tree.EqualFringe ( t1, t2 ) endlet
```

to test if the fringes of the trees described by strings *s1* and *s2* are the same. This assumes that the result of evaluating the module of Figure 1 is named *Tree* in the user's directory.

The design of VIMVAL permits full type checking to be done by the compiler and the *Bind* command. The programmer may declare the types of arguments and results of modules and functions, as shown in the example. The VIMVAL compiler will perform as much checking as it can, reporting conflicts to the user. The *Bind* command will perform the additional checking required to validate the use of free names. Once all free names are bound it is not possible for type errors to occur during module execution.

We plan to employ type inference as developed by Milner [21] in connection with ML (the metalanguage of Edinburgh LCF [13]), and allow most type specifications to be optionally omitted by the programmer [19]. The VIM type-checker will make such type assignments as it can determine and complain if any types remain unknown in a fully bound module. The type-checking module of the VIM Shell will facilitate its task by maintaining a global type list of all types used at the interfaces of modules. In this way, hierarchical composition of program parts is supported with a guarantee of type-correct execution once module binding is completed.

Not requiring that all types be determined at compilation opens the possibility that a module may be bound in two or more contexts, each yielding a different resolution of the types associated with the variable (usually a function variable) being bound. This introduces a useful form of polymorphism of modules and functions. For example, a *Sort* module might specify only that its two arguments be an array of values of an unspecified type named *Item*, and a predicate on pairs of values of type *Item*. The *Sort* module could then be used both to sort arrays of integers and arrays of character strings. The envisioned type-checker supports this by insisting only that separate and consistent type assignments be determined for each context in which a module is bound.

5. Project Status and Plans

VIM is presently being implemented on CADR 29, a Lisp machine built by the MIT Artificial Intelligence Laboratory [18]. Mark I, an initial version of VIM, is a data-driven interpreter for function templates incorporating the mechanisms we have described [22]. It is written in Lisp and implements an instruction set carefully designed to support the VIMVAL language.

Present effort concerns efficient implementation of the heap on the two-level physical memory of the hardware system: semiconductor main memory and disk. Our plan is to treat the main memory as a cache with respect to the large disk (300 megabytes). A small, fixed-size unit of address space called a *chunk* is the unit of memory allocation and the unit of information transmitted to and from the disk. Since each of the principal storage structures (function templates, nested records, arrays, early-completion queues) may be arbitrarily large, each is represented by a tree of chunks. The representation has been designed by Bhaskar Guharoy, who is designing the storage management schemes for VIM [14]. The reference count method of storage reclamation will be used since directed cycles can never arise in the heap, and the method promises to have considerable advantage in a system that supports concurrency and has several levels of physical storage.

Since the usual distinction between active data and files does not exist in VIM, a novel design of data back-up and recovery procedures is required. This is the subject of current research by Suresh Jagannathan [15].

To be able to run programs on VIM, a compiler to translate from VIMVAL into VIM program graphs is needed. For reasonable efficiency, the compiler must perform several important optimizing transformations. For example, it must recognize when tail-recursive implementation of function application can be employed; it must determine when values for record fields are immediately available so use of the early completion scheme may be waived. The design of such a compiler has been worked out and its implementation is in progress based on the existing VAL compiler written in Clu for the DEC 2060. Once the compiler has been completed and tested, it will be rewritten in VIMVAL and installed on VIM.

Eventually the VIM interpreter will be implemented by writing microcode for some appropriate host machine. We expect that once a compact instruction set has been designed and implemented, and efficient disk management schemes developed, VIM will perform competitively with other organizations for shared computer resources. Parallel processing versions of VIM will follow after positive evaluation of the current project.

References

1. Ackerman, W. B. Data Flow Languages. In *AFIPS Conference Proceedings, Volume 48: Proceedings of the 1979 National Computer Conference,* AFIPS, 1979, pp. 1087-1095.

2. Ackerman, W. B. Dataflow Languages. *COMPUTER, 15,2* (Feburary 1982), 15-23.

3. Arvind, and Gostelow, K. P. The U-interpreter. *COMPUTER, 15,2* (Feburary 1982), 42-49.

4. Ashcroft, E.A., and Wadge, W.W. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM 20*, 7 (July 1977), 519-526.

5. Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM 21*, 8 (August 1978), 613-641.

6. Darlington, J., Reeve, M. Alice: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, Association for Computing Machinery, October, 1981, pp. 65-76.

7. Davis, A. L., and Keller, R. M. Dataflow Program Graphs. *COMPUTER, 15,2* (February 1982), 26-41.

8. Dennis, J. B. Programming Generality, Parallelism and Computer Architecture. In *Information Processing 68*, North-Holland Publishing Company, Amsterdam, 1969.

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

10. Dennis, J. B. Data Should Not Change: A Model for a Computer System. Laboratory for Computer Science, MIT, Cambridge, Mass., July, 1981. Submitted for publication

11. Dennis, J. B. An Operational Semantics for a Language with Early Completion Data Structures. In *Formal Descriptions of Programming Concepts*, Springer-Verlag, Berlin, 1981.

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

13. Gordon, Michael, Robin Milner, L. Morris, M.Newey, and C.Wadsworth. A Metalanguage for Interactive Proof in LCF. Proceedings of the Fifth ACM Conference on the Principles of Programming Languages, 1978.

14. Guharoy, B. Memory management in a Dynamic Data Flow Computer System. Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., expected August, 1984.

15. Jagannathan, S. . Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., expected February, 1985.

16. Johnsson, T. The G-Machine: An Abstract Machine for Graph Reduction. Programming Methodology Group, Chalmers University of Technology, Goteborg, Sweden, 1983.

17. Keller, Robert, Gary Lindstrom, and Suhas Patil. A loosely-coupled applicative multi-processing system. *Proc. of the National Computer Conference, ACM* (June 1979), 613-622.

18. Knight, T. F., et al. CADR. *Memo No. 528, A. I. Laboratory, M. I. T.* (March 1981).

19. Kuszmaul, B. Type Inference for VimVal (tentative). Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., May, 1984. Bachelor's thesis

20. McGraw, J. R. The VAL Language: Description and Analysis. *ACM Transactions on Programming Languages and Systems 4*, 1 (January 1982), 44-82.

21. Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348-375.

22. Stoy, J. E. VIM--A Dynamic Dataflow Implementation of VAL. In preparation.

23. Turner, D. A. A New Implementation Technique for Applicative Languages. *Software-Practice and Experience 9* (December 1977), 31-49.

24. Turner, D.A. The Semantic Elegance of Applicative Languages. Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, Association for Computing Machinery, October, 1981, pp. 85-98.