An Abstract Implementation for
Concurrent Computation With Streams

by

Jack Dennis
Ken Weng

This paper will appear in the Proceedings of the 1979 International
Conference on Parallel Processing.

July 1979
AN ABSTRACT IMPLEMENTATION
FOR
CONCURRENT COMPUTATION WITH STREAMS(a)

Jack B. Dennis
Ken K.-S. Weng
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- This paper is a contribution toward
developing practical general-purpose computer systems
embodying data flow principles. We outline a hardware
structure capable of high concurrency and present an
abstract model of data flow program execution which could
be implemented within the proposed hardware structure.
Our abstract model supports a user programming language
that includes recursive function modules and provides
streams of values for inter-module communication.

A Simple Value-Oriented Language

Our textual language departs from conventional
languages in several ways. There is no notion of sequential
control flow and there are no explicit primitives for
introducing parallelism. The concurrency of a computation is
determined by the data dependency within the program
rather than by explicit creation of concurrent processes.

The language is value-oriented in the sense that each
syntaxic unit defines a mathematical function that maps
input values into result values; there are no side effects or
other spurious interactions in the evaluation of expressions.

The language does not have the notion of memory
locations or variables commonly found in conventional
sequential programming languages; instead names are used
to denote values defined by expressions in much the same
way as in mathematics. With value-oriented semantics, it is
natural to write programs in a form that exhibits the inherent
concurrency of an algorithm. The data types of the
language(a) are integer, real, boolean, character-string,
structure, and procedure. We shall call these data types
simple data types. The operations for types integer, real,
boolean, and character-string are the usual operations and
need no comment. The operations for values of type
structure are defined below. The only operation for
procedure values is procedure application.

The syntax of the language is given in Fig. 1. A
procedure consists of a set of procedure definitions
followed by an expression. A procedure definition is of the
form

P = procedure (a1:T1,...,am:Tm) yields R1,...,Rn;
(procedure def)
.
.
.
(procedure def)
(expression)
end P;

(a) The language described here is closely related to the
language called VAL in development at MIT [3].

Specifically, we introduce a value-oriented language
and discuss representation of its semantics by translation
into recursive data flow schemas [9]. We sketch an
operational semantics (formal interpreter) for these data
flow schemas and outline the structure of a hardware
system capable of highly concurrent execution of
value-oriented programs. A more detailed and complete
presentation of this work is given in the thesis of Weng
[17].

(a) This research was supported in part by the National
Science Foundation under research grant MCS75-04050
AO1 and in part by the Lawrence Livermore Laboratory of
the University of California under contract 8845463.
Notation:

\[(E)\] means \(E\) | \([E]\) |

\( (E)^* \) means \( (E) \) | \( (E)^* \) |

\( (E)^+ \) means \( (E)^* \) - empty

\( \langle \text{program} \rangle ::= \langle \text{program} \rangle \langle \text{expression} \rangle \langle \text{end} \rangle \)

\( \langle \text{procedure def} \rangle ::= \langle \text{name} \rangle \langle \text{procedure} \rangle \langle \text{input list} \rangle \)

\( \langle \text{yield} \rangle \langle \text{output list} \rangle ; \)

\( \langle \text{procedure def} \rangle ; \langle \text{expression} \rangle \langle \text{end} \rangle \langle \text{name} \rangle \)

\( \langle \text{input list} \rangle ::= \{ \langle \text{type declaration} \rangle \} \)

\( \langle \text{type declaration} \rangle ::= \langle \text{name} \rangle \langle \text{type} \rangle \)

\( \langle \text{output list} \rangle ::= \{ \langle \text{type} \rangle \} \)

\( \langle \text{expression} \rangle ::= \langle \text{primitive expression} \rangle \)

\( | \langle \text{expression} \rangle^* \)

\( | \langle \text{let-block expression} \rangle \)

\( | \langle \text{conditional expression} \rangle \)

\( | \langle \text{application expression} \rangle \)

\( \langle \text{let-block expression} \rangle ::= \langle \text{let} \rangle \langle \text{type declaration} \rangle ; \langle \text{name def} \rangle ; \langle \text{expression} \rangle \langle \text{end} \rangle \)

\( \langle \text{name def} \rangle ::= \langle \text{name} \rangle \langle \text{expression} \rangle \)

\( \langle \text{conditional expression} \rangle ::= \langle \text{if} \rangle \langle \text{expression} \rangle \langle \text{then} \rangle \langle \text{expression} \rangle \langle \text{else} \rangle \langle \text{expression} \rangle \langle \text{end} \rangle \)

\( \langle \text{application expression} \rangle ::= \langle \text{name} \rangle \langle \text{expression} \rangle \)

\( \langle \text{primitive expression} \rangle ::= \langle \text{expression} \rangle \langle \text{primitive operation} \rangle \langle \text{expression} \rangle \)

\( | \langle \text{primitive operation} \rangle \langle \text{expression} \rangle \)

\( | \langle \text{name} \rangle \)

\( | \langle \text{constant} \rangle \)

\( \langle \text{simple data type} \rangle ::= \langle \text{integer} \rangle \langle \text{real} \rangle \langle \text{boolean} \rangle \langle \text{character-string} \rangle \langle \text{structure} \rangle \)

\( \langle \text{type} \rangle ::= \langle \text{simple data type} \rangle \langle \text{stream of} \langle \text{simple data type} \rangle \)

---

Figure 1. Syntax of the language

This defines a procedure \(P\) that requires \(n\) input values \(x_1, \ldots, x_n\) of types \(T_1, \ldots, T_n\) respectively. The names \(x_1, \ldots, x_n\) must be distinct and can appear free in \(E\). The evaluation of the procedure yields an ordered set of values \(R_1, \ldots, R_n\) resulting from \(E\).

Each expression denotes an ordered set (n-tuple) of values whose arity is \(n\). We give a recursive definition of the arity \(A(E)\) of each of the five types of expressions as follows:

\[
A(\langle \text{primitive expression} \rangle) = 1
\]

\[
A(\langle \text{expression} \rangle) = A(\langle \text{expression} \rangle) + 1
\]

\[
A(\langle \text{let-block expression} \rangle) = A(\langle \text{let} \rangle \langle \text{definitions} \rangle \langle \text{expression} \rangle \langle \text{end} \rangle)
\]

\[
A(\langle \text{name} \rangle) = A(\langle \text{expression} \rangle)
\]

For a \(\langle \text{procedure def} \rangle\) to be correct, the arity of the expression which is its body must match the number of result types specified in its \(\langle \text{output list} \rangle\).

Often it is convenient to introduce names for expressions because they are common subexpressions of larger expressions. The let-block expression is used for introducing names such that each name stands for an
expression of arity one. A let-block expression is of the form:

\[
\begin{align*}
\text{let} & \quad \{ (\text{type declaration}) \} ; \\
& \quad (\text{name-list}_1) = (\text{exp}_1) ; \\
& \quad \vdots \\
& \quad (\text{name-list}_k) = (\text{exp}_k) ; \\
& \quad \text{in} \quad (\text{exp}) \quad \text{end}.
\end{align*}
\]

The names in type declarations of a let-block are local names meaningful only within the block; these names must be distinct from each other and may appear free in \(\text{exp}_1, \ldots, \text{exp}_k\), and \(\text{exp}\). Name conflicts in nested let-blocks are resolved by the scope rule that inner definitions take precedence over outer definitions.

We require that the number of names in a name-list be equal to the arity of the expression appearing on the right hand side of the equality sign. The value of a name in a name-list is the value of the corresponding expression appearing on the right hand side of the equality sign, and must be of the type specified by the type declaration. The value of a let-block expression is the value of \(\text{exp}\).

A conditional expression is of the form:

\[
\begin{align*}
\text{if} \quad (\text{exp}_1) \quad \text{then} \quad (\text{exp}_2) \quad \text{else} \quad (\text{exp}_3) \quad \text{end.}
\end{align*}
\]

The expression \(\text{exp}_1\) is a boolean value of arity one. The expressions \(\text{exp}_2\) and \(\text{exp}_3\) have the same arity and the corresponding value in each expression must be of the same type. The value of a conditional expression is the value of \(\text{exp}_2\) if \(\text{exp}_1\) is the boolean value \text{true}; otherwise it is the value of \(\text{exp}_3\).

A procedure application expression is of the form:

\[
\begin{align*}
\text{P} \quad (\text{exp}) \quad ;
\end{align*}
\]

where the expression \(\text{exp}\) has the same arity as the number of input values required by the procedure \text{P} and the type of each value matches that of the input specification. The result of the procedure application is an expression of the arity and types defined by the yield clause of the procedure heading.

As a simple example of a program in our value-oriented language, Fig. 2 shows a procedure that defines a parallel computation of the factorial function.

**Data Structures**

For the purpose of the present exposition, we will introduce a simple but very general data structure type. A data structure can be either \text{nil} which denotes the structure having no components, or a structure having \(n\) component values \(v_1, \ldots, v_n\) whose selector names are respectively \(s_1, \ldots, s_n\). The selectors are either character strings or integers and each selector name must be different from all others in the same data structure. We represent such a structure value by the notation

\[
(s_1 : v_1, \ldots, s_n : v_n).
\]

The operations on data structures are defined below, where \(d\) and \(d'\) are data structures, \(s\) is a selector name, and \(c\) is a value of any type:

1. \text{create}( )
   
   The \text{create} operation yields the \text{nil} data structure.
2. \text{append}(d, s, c)
   
   The result is a data structure \(d'\) which is identical to \(d\) except that the \(s\) component is \(c\) regardless of whether \(d\) already contains a component with selector name \(s\).
3. \text{delete}(d, s)
   
   The result is a data structure \(d'\) which does not have an \(s\) component.
4. \text{select}(d, s)
   
   If \(d\) has an \(s\) component, the result is the value of that component. Otherwise, the result is the value undefined.
5. \text{nil-structure}(d)
   
   This is a predicate whose value is \text{true} if \(d\) is \text{nil}; otherwise its value is \text{false}.

Notice that the effects of

\[
\text{delete}(d, s)
\]

and

\[
\text{append}(d, s, \text{nil})
\]

are different, since the the \text{delete} operation would remove the component \((s, d')\) while the \text{append} operation would replace it with \((s, \text{nil})\). It should be mentioned that an array
reverse = procedure (x : structure) yields structure;

if nil-structure (x) then x else
  let left, right : structure;
  left = reverse( select(x, "r") );
  right = reverse( select(x, "l") );
  in append( append( create( ), "r", left ), "l", right )
end
end reverse;

Figure 3. reverse

is simply a data structure whose selector names are all integers.

The data structure operations are illustrated by the recursive procedure "reverse" in Fig. 3, which interchanges the role of selector names 'l' and 'r' in a given data structure of arbitrary depth.

Streams

A stream is a sequence of values, all of the same type, that are passed in succession, one-at-a-time between program modules.

The use of streams of data in programming is an alternative way of expressing computations that have conventionally been expressed as coroutines or a set of cooperating processes. For example, a compiler may be organized into phases which are implemented as a set of coroutines [6].

The operations on values of type stream of T are defined below where a and a' are streams, and c is a value of type T.

1) [ ]
   The result is the empty stream which is the sequence of length zero.
2) cons (c, a)
   The result is a stream a' whose first element is c and whose remaining elements are the elements of the stream a.
3) first (a)
   The result is the value c which is the first element of a. If a is empty, the result is undefined.
4) rest (a)
   The result is the stream left after removing the first element of a. If a = [ ], the result is undefined.
5) empty (a)
   The result is true if a = [ ], and is false otherwise.

prime_generator = procedure (n : integer) yields stream of integer;

generate = procedure (i, n : integer) yields stream of integer;

if i < n then [ ]
  else cons ( i, generate( i+1, n ) ) end;
end generate;

sieve = procedure (a : stream of integer) yields stream of integer;

if empty (a) then [ ]
  else let x : integer,
  a_2, a_3 : stream of integer;
  x, a_2 = first (a), rest (a);
  a_3 = delete (x, a_2);
  in cons (x, sieve(a_3)) end;
end sieve;

delete = procedure (x : integer,
  a : stream of integer) yields stream of Integer;

if empty (a) then [ ]
  else let y : integer,
  a_2, a_3 : stream of integer;
  y, a_2 = first (a), rest (a);
  a_3 = delete (x, a_3);
  in if divide (x, y) then a_3
  else cons (y, a_3) end;
end:
end delete;

sieve (generate (2, n));
end prime_generator;

Figure 4. A Prime Number Generator

The following identity is satisfied by the stream operations:

if empty (a) then s = [ ]
  else s = cons( first (a), rest (a) ) end

The problem of generating all prime numbers less than a given integer n is a good example of the use of streams in constructing a modular program so as to expose many independent actions for concurrent execution. The sieve of
Eratosthenes expressed in our textural language is presented in Fig. 4. The procedure "generate" produces the sequence of successive integers beginning with 2. This stream is processed by "sieve" to remove nonprime elements. Procedure "sieve" operates by taking the first element of its input and removing all multiples of the first element (using "delete") and applying "sieve" recursively to the remaining elements. (The first use of stream concepts for the prime number sieve, as far as we know, was in [10]. It seems the example has been discovered independently by several authors.)

Data flow schemas

A data flow schema is an operational model of concurrent computation. The form of schemas used here derives from the work of Dennis and Fosseen [9] and Dennis [7]. A data flow schema is a directed graph composed of nodes called actors and arcs connecting them. An arc pointing to an actor is called an input arc of the actor; and an output arc is an arc emanating from the actor. Each actor has an ordered set of input arcs and output arcs. There are five types of actors: link, operator, switch, merge, and sink. The five types of actors are shown in Fig. 5. An (m, n) data flow schema must have m links which do not have input arcs, and n links not having output arcs. These links are respectively called input links and output links of the (m, n) schema. Further, we require that the schema must be proper in the sense that all other actors must have the required arcs of its actor type, and each arc must be connected at both ends.

Firing Rules

Execution of an (m, n) schema advances it from one configuration to another through the firing of some actor that is enabled. The firing rules for the principal actor types are specified in Fig. 6. A necessary condition for any actor to be enabled is that each output arc does not hold a token. An actor is enabled when a token is present on each input arc with the exception of a merge actor. The firing of an actor causes the tokens to be absorbed from the input arcs and completes by placing a token on each of the output arcs. The values of the output tokens are functionally related to the values of the input tokens. A link simply replicates the value received and distributes it to the destination actors indicated by output arcs. The effect of firing an operator is to apply to the inputs \( v_1, \ldots, v_m \) the function associated with the operation name written inside the operator to yield the outputs \( v'_1, \ldots, v'_n \). The switch and merge are used for controlling the flow of tokens. A switch requires a data input and a control input which is a boolean value. The firing of a switch replicates the input token on one of the output arcs according to the boolean control value. The arrival of a token on either input arc enables a
merge, and upon firing, a token conveying the same value is placed on the output arc. The behavior of a merge is inherently nondeterministic: when two input tokens reside on the input arcs, the firing rule does not specify in which order the output tokens will be generated. A sink absorbs the input tokens upon firing and places a special token signal on the output arc. The purpose of a sink actor is to absorb unwanted values; the signal output token is necessary for the implementation of schema application to be described.

The set of functions commonly associated with an operator includes the scalar arithmetic operations and constant functions.

**Well Formed Data Flow Schemes**

Unrestricted use of actors in data flow schemes is undesirable since an arbitrary interconnection of these actors may form a schema which deadlocks or has nondeterminate behavior. Because these properties are undesirable for reliable programming we choose a subclass of schemes which will satisfy the needs of programming.

An \((m, n)\) well formed data flow schema is an \((m, n)\) data flow schema formed by any acyclic composition of component data flow schemes, where each component is either a link, a sink, an operator, or a conditional subschema.

**Fig. 7** is an example of a conditional schema which computes the value of the expression

\[
\text{if } a > b \text{ then } a + b \text{ else } b - 3
\]

Here, the trig output provides a completion signal indicating that the sink actor has absorbed the unused copy of \(a\). The structure of a conditional schema corresponds in an obvious way to conditional expressions.

**The Apply Actor**

The class of well formed data flow schemes cannot express program features such as procedures, procedure applications, and iterations. We introduce an actor apply whose meaning is explained in Fig. 8. The first input to an apply actor is a token associated with an \((m, n)\) well formed data flow schema. An apply actor is enabled when a token is present on each input arc. The effect of firing an apply actor is to replace the actor with the specified \((m, n)\) schema as shown in the figure. The \((m, n)\) schema replacing the apply actor may itself contain apply actors, allowing recursion to be expressed.

We have not included structures of data flow schemes which correspond to language constructs such as while loops in Algol 60 or Do statements in Fortran. Such structures necessarily involve cyclic connections of actors which do not correspond to actual data dependencies, and introduce unnecessary delays. Furthermore, the semantics of cyclic schemes is more complicated, since issues of safety and liveness must be dealt with. We choose to support these language features in the equivalent form of recursive application of data flow schemes. This allows simultaneous execution of instances of a data flow schema which correspond to successive iterations of a while loop.

An example of the use of apply actors is given in Fig. 9. This recursive schema implements the "reverse" function stated earlier in Fig. 8. The input link actor labeled trig is an input link whose function is to trigger those actors that generate constants; in this case the create actor that produces the empty data structure.

The apply actor presented requires that all input values be present on the input arcs to become enabled. A language implemented in terms of the apply actor will have "call by value" semantics, that is, the result of application is well defined only when the computations producing arguments to the procedure all terminate. This is in contrast with a more general form of procedure application which allows procedure application to begin even though computation of some arguments is not complete.

**Data Flow Processor**

The structure of a data flow processor suitable for supporting execution of recursive data flow schemes is shown Fig. 10. It consists of six subsystems: Functional Units, Structure Controller, Execution Controller, the Arbitration and Distribution Networks, and the Packet
Memory. The Execution Controller fetches instructions and
operands from the Packet Memory and forms them into
operation packets. Each operation packet is passed to the
Arbitration Network for transmission to an appropriate
Functional Unit if a scalar operation is called for, or to the
Structure Controller for the data structure operations
create, append, and select. Instruction execution in the
Structure Controller and Functional Units generate result
packets which are sent through the Distribution Network to
the Execution Controller where they will join with other
operands to activate their target instructions. How this is
done is explained in greater detail in the next section.

The Packet Memory holds the collection of data
structures as a collection of items each being a one-level
data structure having scalar values and unique identifiers of
other items as its components [6]. This collection of items
represents an acyclic directed graph where each arc
corresponds to a unique identifier component of the item
representing its origin node. The Packet Memory maintains
a reference count for each item and recycles physical
storage space as items become inaccessible.

Data structures held in the Packet Memory have three
roles in the execution of data flow schemes: (1) as
operands for the data structure operations implemented by
the Structure Controller; (2) as procedure structures that
have as components the instructions of a data flow
procedure; and (3) activation records which hold operand
values for instructions waiting for their enabling condition
to be satisfied.

Although the Execution Controller, Structure Controller
and the Packet Memory are shown in Fig. 10 as single units,
we imagine that each is in fact a collection of many identical
units. For example, the Packet Memory subsystem would
consist of separate systems, each holding all items whose
unique identifiers belong to a well defined part of the
address space of unique identifiers. The Execution
Controller subsystem would consist of identical modules
each of which would serve a distinct subset of procedure
activations.

The concept of a Packet Memory System was
introduced in [6], and the design issues for these systems
and the Structure Controller have been studied in [1, 2].

Implementation of Data Flow Schemes

Procedure Structures

A data flow scheme is represented in the machine by a
kind of data structure called a procedure structure
illustrated in Fig. 11a. A procedure structure corresponding
to a data flow scheme of n actors is a data structure having
n components with integer selector names from 1 to n
assigned to the actors. Each component, called an
Instruction, is an encoding of an actor and its output arcs.
The components of an instruction include an operation field which defines the function performed by the actor, and destination fields D1, ..., Dp corresponding to p output arcs. Each destination field has three subcomponents: the inst component is the integer selector name of the destination instruction; the arc component is an integer designation of an input arc of the destination; and the count component is the number of operand values required by the destination instruction.

**Activation Records**

Since multiple instances of the same schema may be concurrently active in a computation, each activation (an instance of procedure execution) is represented by a separate activation record as shown in Fig. 11b. Each actor in an activation is uniquely identified by the tuple (A, i), where A is a uid allocated for the activation record and i is the integer assigned to the actor in the procedure structure. A token of value v on the k-th input arc of an actor (A, i) corresponds to a result packet that carries the information (A, i, k, v, count), where "count" is the number of tokens (operands) required for the enabling of the actor.

Enabling of an actor is detected by checking the number of result packets having arrived at the operand record -- the 1 component of the activation record A against the count in the result packet. The detection of enabling is a function of the Execution Controller and the Packet Memory that store activation records. Upon enabling of actor instance (A, i), the instruction of the actor is fetched from the 1 component of the procedure structure. The following section describes how activation records might be manipulated.

An activation record has components with integer selectors for operand records and an additional "text" component that is the procedure structure for the activation. (In our implementation, this component is shared by other activations of the same schema.) An operand record may have as many integer subcomponents as input arcs of an actor, and also contains an "arrived" subcomponent indicating the number of arrived result packets. Since an activation record stores values of arrived result packets in its components, operations on an activation record modify its components. These operations are defined as follows:

![Diagram](image)
(1) create-activation( P )

This returns the uid of a new activation record having P as its "text" component, but no other components.

(2) insert( A, i, k, v )

The insert operation adds the value v as the k-th operand of the i-th instruction in activation record A. In addition, the "err" component of the operand record is incremented by one. To handle the first operand value to arrive, a missing "err" component is interpreted as having the value zero.

(3) remove( A, i )

This operation releases the i component of A; and is performed by the Execution Controller once it has generated the operation packet for actor instance (A, i).

(4) free( A )

This operation releases the entire activation record A by means of a command packet sent to the Packet Memory.

For each arriving result packet ( A, i, k, count, v ) the Execution Controller performs the operation insert( A, i, k, v ) and tests the updated value of the "count" component against the "count" field of the result packet. If the values are equal, the instruction is fetched from the Packet Memory and stored, together with the operand record, to construct an operation packet which is delivered to the Arbitration Network. The i component of activation record A is then released.

Procedure Activation

Our implementation of the apply actor is illustrated in Fig. 12. The apply actor is replaced by the code diagrammed in Fig. 12b, and the applied graph F is augmented as in Fig. 12c. Here we use the notations

![Diagram](image)

and

![Diagram](image)

to mean insert( A, i, 1, v ). The new actors extr-uid, const-ret and distribute will be explained below.

This implementation assumes the actors in each recursive scheme are numbered according to this rule:

(1) Input link actors are numbered 1, ..., m.
(2) The link actors that receive the n-tuple of values resulting from a schema application are numbered J + 1, ..., J + n for some integer J.
(3) A link actor numbered 0 receives a packet (A, J, n) containing the information needed to construct result packets for returning values resulting from procedure execution.
(4) The remaining actors may be numbered arbitrarily.

![Diagram](image)

Figure 12. Implementation of apply.

The implementation scheme works as follows: The create-act actor produces the uid A' of a new activation record containing "text" component F' and passes it to the insert actors associated with input value v1, v2, ... The actors cause result packets of the form (A', i, 1, 1, vj) to be generated which initiate execution of the new activation of F'. At the same time, the extr-uid and const-ret actors form the return value (A, J, n) and send it to link 0 of schema F'. Once result values v1, v2, ..., vn have been produced, the distribute and insert actors of F' generate result packets of the form (A, J + 1, 1, vj) which deliver result values to the calling scheme. The free actor then releases the activation record, and its uid A' is returned to the pool of free uid's managed by the Packet Memory.

Implementation of Stream Actors

In the implementation streams are represented as data structures. A stream is a data structure having an "f" component which is the first element of the stream, and an "r" component which is the data structure representing the
rest of the stream. The empty stream is represented by nil. Operations on streams become operations on structure values; thus first(a) and rest(a) are implemented by select(s,"a") and select(s,"r"), respectively.

We wish to make it possible for a stream to be processed by consuming modules while further stream elements are generated concurrently. To provide for this behavior, we must augment our concept of data structures so a data structure may be accessed before it is entirely constructed. We use the concept of holes which is based on the work of Henderson [11] who used the term "token". Our idea is related to but different from the idea of "suspensions" discussed by Friedman and Wadler [10].

The idea is embodied in the implementation of the cons operation described in Fig. 13. Here the create-hole and write-hole actors are special data structure operators defined as follows:

A create-hole actor returns a new H allocated from the data structure address space. The free node is called a hole in that it has two states: filled and unfilled. In the unfilled state, all data structure operations on the hole are queued except the write-hole operation. Upon completion of the write-hole(H,v) operation, the hole H changes its state to filled and contains the value v. All previously queued and subsequent operations on H are processed without further delay; a subsequent write-hole operation on H is illegal.

To illustrate the concurrency provided by this implementation of streams, consider the recursive schema

\[ s' = \text{cons}(v, s) \]

shown in Fig. 14 for the "sieve" procedure of the prime number generator. Note that the output of the top activation of "sieve" will be a data structure containing the first element of the result stream and a hole waiting to be filled in with the data structure generated by the recursive activation of "sieve". In this implementation each higher activation of "sieve" may be released as soon as it has completed its work (i.e., its hole has been filled), leaving the remaining work to be finished by deeper activations of the code.

**Remarks**

The concept of stream has appeared in many forms [5, 12, 14, 15]. One of the earliest papers that discussed streams as a programming feature was an unpublished paper by McIlroy [16]. Despite the conceptual elegance of streams, programming has not yet departed from the sequential notion of coroutines and process synchronization.
primitives. Recent interest in concurrent programming languages and processors have motivated several other authors to investigate the feasibility of implementation of streams and related concepts of data structures with holes or with suspensions [4, 10, 13].

References


