AN APPROACH TO FAIR APPLICATIVE MULTIPROGRAMMING*

by

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 84

AN APPROACH TO FAIR APPLICATIVE MULTIPROGRAMMING

DANIEL P. FRIEDMAN

DAVID S. WISE

APRIL, 1979

*To appear in: G. Kahn and R. Milner (eds.), Proc. of Intl. Symp. on Semantics of Concurrent Computation, Berlin, Springer (1979).

Research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS75-06678 A01 and MCS77-22325.

An Approach to Fair Applicative Multiprogramming

Daniel P. Friedman
David S. Wise
Computer Science Department
Indiana University
Bloomington, IN 47405/USA

Abstract

This paper presents a brief formal semantics of constructors for ordered sequences (*cons*) and for unordered multisets (*frons*) followed by a detailed operational semantics for both. A multiset is a generalization of a list structure which lacks order *a priori*; its order is determined by the *a posteriori* migration of computationally convergent elements to the front. The introductory material includes an example which demonstrates that a multiset of yet-unconverged values and a timing primitive may be used to implement the scheduler for an operating system in an applicative style. The operational semantics, given in PASCAL-like code, is described in two detailed steps: first a uniprocessor implementation of the *cons*/*frons* constructors and the *first*/*rest* probes, followed by an extension to a multiprocessor implementation. The center of either implementation is the EUREKA structure transformation, which brings convergent elements to the fore while preserving order of shared structures. The multiprocessor version is designed to run on an arbitrary number of processors with only one semaphore but makes heavy use of the **sting** memory store primitive. **Stinging** is a conditional store operation which is carried out independently of its dispatching processor so that shared nodes may be somewhat altered without interfering with other processors. An appendix presents the extension of this code to a "fair" implementation of multisets.

Introduction

This paper is directly motivated by a practical implementation of a constructor function for applicative multiprogramming [2]. The new constructor, dubbed *frons*, offers a new perspective on the problem of synchronizing inherently asynchronous computation, by allowing asynchronous processes to be assembled into an unordered structure which behaves like an ordered structure upon access. The order is determined by their

relative order of convergence.

The demands of a fair implementation (presented only as the appendix) for an environment of asynchronous processors have been difficult to master. The solution has prompted the invention of new synchronization primitives for imperative (von Neumann-style) programming, so that this implementation is not even practical on current hardware. In particular, we present here the programs which prompted the invention of the **sting** conditional store instruction [3], whose implementation is tractable.

What follows, then, is an operational introduction to the *frons* (and *cons*) constructor using the **sting** primitive for PASCAL-like languages. There are six sections: Formal Semantics; Examples (which presents a scheduler for an operating system); a section on implementing the constructors and the probing functions, *first* and *rest*, which depend on EUREKA. EUREKA is the centerpiece of the operational semantics, because *it* is responsible for uniformly evaluating several suspended evaluations and transforming the unordered structures into "slightly ordered" ones while preserving the formal semantics. The bulk of the paper appears in the last three sections: a uniprocessor EUREKA; an extension of that to a multiprocessor EUREKA; and finally some conclusions.

Formal Semantics

The *frons* constructor is introduced elsewhere [2] where its semantics and use are developed in greater detail. (The Latin word "frons" means "leafy branch" and is motivated by the definition of *Fern* there.) We present its definition in a form similar to LISP's axioms [6] before proceeding to a formal structure semantics.

$$\text{first:NIL} = \perp = \text{rest:NIL}. \tag{1}$$

Let z = cons:<x y> then

$$\text{atom:z} = \textit{false} = \text{null:z}; \tag{2}$$
$$\text{first:z} = \text{x}; \tag{3}$$
$$\text{rest:z} = \text{y}. \tag{4}$$

Let z = frons:<x y> then we define the same primitives

$$\text{atom:z} = \textit{false} = \text{null:z}; \tag{5}$$

$$\text{first:z} = \begin{cases} \text{x, if } \text{x} \neq \perp ; \\ \text{first:y, if first:y} \neq \perp ; \\ \text{x, if y = NIL ;} \end{cases} \quad \begin{cases} \text{where the choice made} \\ \text{must be the same as} \\ \text{that made for } \textit{rest}. \end{cases} \tag{6}$$

$$\text{rest:z} = \begin{cases} \text{y, if } \text{x} \neq \perp ; \\ \text{frons:<x rest:y>, if first:y} \neq \perp ; \\ \text{y, if y = NIL ;} \end{cases} \quad \begin{cases} \text{where the choice made} \\ \text{must be the same as} \\ \text{that made for } \textit{first}. \end{cases} \tag{7}$$

The binding of z to a particular structure requires that the choices

made in (6) and (7) must be the same when both x and first:y converge. This semantics is slightly different from [2] because we have included a third alternative; its effect is that cons:<x NIL> = frons:<x NIL>; singleton unordered structures evaluate to singleton lists.

Just as the purpose of *cons* is to construct ordered structures, so also does *frons* construct unordered structures. Since the content of either form of data structure may be suspended, we may use behavior of these postponed evaluations to advantage in case one or more of them might diverge. In the definitions above we allow that the convergent elements make their way to the front of the unordered structures, or *multisets*, and once they arrive they stay! The structure therefore behaves like a sequence upon probing.

Definition: STRICTIFY(X,Y) = Y if $X \neq \perp$;

$$\text{AMB}(X,Y) = \begin{cases} X, & \text{if } X \neq \perp; \\ Y, & \text{if } Y \neq \perp. \end{cases}$$

The function AMB is from McCarthy [6] where it is introduced as a non-deterministic choice operation. We shall only use it embedded *within* a structure to determine order, so that we only calculate AMB once--as in called-by-delayed value [8]---for any structure.

We define the set $\mathcal{D}$ of all computational structures to include the set of atomic items A (and whatever semantics one likes for them) and the set of structures S which are quadruples (or trivially NIL).

$$\mathcal{D} = A \cup S;$$

$$S = \{\text{NIL}\} \cup (\mathcal{D}^+ \times S^+ \times \{\text{TRUE},\text{FALSE}\}^+ \times \omega);$$

where ω is the set of natural numbers; the fourth element is only used as an index on the quadruple whose effect on semantics is to guarantee uniqueness of structure. In any implementation no two quadruples will have the same index (analogous to memory address). Thus, the domain equations for our primitives follow:

null: $S \to \{true, false\}$;

cons: $\mathcal{D}^+ \times S^+ \to S$;

frons: $\mathcal{D}^+ \times S^+ \to S$;

first: $S \to \mathcal{D}^+$;

rest: $S \to S^+$;

strictify: $\mathcal{D} \times \mathcal{D}^+ \to \mathcal{D}^+$.

In the following definition the occurrence of "i $\in \omega$" denotes a

new integer which has not occurred in any other quadruple. As mentioned above, for any structures indexed by such an i as a fourth element, an AMB expression as the third element is evaluated *at most once* and its evaluation is delayed until accessed [1] (i.e. call-by-need, call-by-delayed value or lazy evaluation is implied [9,8,4]).

Definition:

$$I[\![\text{null:f}]\!] = \begin{cases} \text{TRUE, if } I[\![f]\!] = \text{NIL, the trivial element of } S; \\ \text{FALSE, if } I[\![f]\!] \text{ is a quadruple in } S. \end{cases}$$

$I[\![\text{cons:pair}]\!] = (I[\![1\text{:pair}]\!], I[\![2\text{:pair}]\!], \text{TRUE}, i)$ where $i \in \omega$.

```
I[[frons:pair]] = (I[[1:pair]],
                   I[[2:pair]],
                   AMB (STRICTIFY(I[[1:pair]], TRUE),
                        STRICTIFY(I[[first:2:pair]], FALSE)),
                   i) where i ∈ ω.

I[[first:f]] = FIRST(I[[f]]) where
                 FIRST(NIL) = ⊥;
                 FIRST((u, v, TRUE, j)) = u;
                 FIRST((u, v, FALSE, j)) = FIRST(v); and
                 FIRST((u, NIL, b, j)) = u.

I[[rest:f]] = REST(I[[f]]) where
                REST(NIL) = ⊥;
                REST((u, v, TRUE, j)) = v;
                REST((u, v, FALSE, j)) =
                  (u,
                   REST(v),
                   AMB(STRICTIFY(u, TRUE),
                       STRICTIFY(FIRST(REST(v)), FALSE)),
                   i) where i ∈ ω; and
                REST((u, NIL, b, j)) = NIL.
```

$I[\![\text{strictify:pair}]\!] = I[\![2\text{:pair}]\!]$, if $I[\![1\text{:pair}]\!] \neq \perp$.

Notation:

$I[\![1\text{:f}]\!] = I[\![\text{first:f}]\!]$;

$I[\![2\text{:f}]\!] = I[\![\text{first:rest:f}]\!]$;

$I[\![<>]\!] = \text{NIL} = I[\![\{\}]\!]$;

$I[\![<x_1\ x_2\ \ldots\ x_k>]\!] = I[\![\text{cons:}<x_1\ <x_2\ \ldots\ x_k>>]\!]$;

$I[\![\{x_1\ x_2\ \ldots\ x_k\}]\!] = I[\![\text{frons:}<x_1\ \{x_2\ \ldots\ x_k\}>]\!]$;

$I[\![\{x_1 \mid x_2\ \ldots\ x_k\}]\!] = I[\![\text{cons:}<x_1\ \{x_2\ \ldots\ x_k\}>]\!]$.

Examples

Two examples will help explain the applications of *frons* and multisets. In both cases correctness proofs will follow known recursion-induction techniques; a stronger correctness will follow when fairness is established.

The first example demonstrates the power of encapsulating the choice (or AMB operator) within the structure, because a familiar function definition does not change from its traditional form in order to introduce nondeterminism. The conventional operator for taking the disjunction of arbitrarily many arguments is *or*.

```
or:disjuncts ≡
  if null:disjuncts then false
  elseif first:disjuncts then true
  else or:rest:disjuncts  .
```

If *or* is applied to the sequence of truth values $\langle b_1\ b_2\ \ldots\ b_k \rangle$ where $B = \{true, false\}$ and $b_i \in B^+$ then its value is the desired value in B unless $\perp$ precedes *true* in that list. If *or* is applied to the multiset of values $\{b_1\ b_2\ \ldots\ b_k\}$, however, its result will be in B whenever *true* is in that list of values or whenever $\perp$ is not, regardless of the order. Thus the multiset as an argument structure implements the *symmetric or* [7] without redefinition of the function.

As an intermediate example we consider the conversion of a sequence of (perhaps not yet converged) elements into a multiset. In the scheduler example below the need arises for the function *scramble* which introduces disorder into the structure of potential solutions of (what is stated as) a sequence of jobs.

```
scramble:seq ≡
  if null:seq then {}
  else frons:<first:seq scramble:rest:seq>.
```

The second example is the timing of independent simultaneous processes as a distributed processor might do for scheduling purposes. The problem is to take a sequence of (unevaluated) expressions as an argument and to return those (still not completely evaluated) which do not converge after t units of "time". We shall develop its solution as a series of functions, the proof of which would establish the validity of the entire solution. With the intermediate example we may anticipate that the input argument, *exprlist*, may as well be a multiset.

The "function" *clock* is a constant-valued function whose implementation, like *strictify* (in our call-by-need environment) cannot be optimized; we require it to consume t units of "time" before it converges:

```
clock:t ≡
  if zerop:t then ALARM
  else clock:pred:t .
```

(zerop:n = *true* if n = 0; pred:n = n-1 if n > 0.)

We do not suggest that the *clock* function need be perfectly accurate any more than the peripheral hardware clock on actual computers will interrupt at future, predictable machine cycles. We do assume that for relatively larger settings of its argument, it will converge in times asymtotically proportional to those settings. Such a *clock* will be added to the multiset of unconverged values.

We partition the resulting multiset by separating all elements up to the occurrence of ALARM from the remainder. If the implementation of multisets is fair then we may be assured that its prefix consists of the values of the expressions which converged within the time-limit set.

```
partition:M ≡
  if same:<first:M ALARM> then < <> rest:M >
  else buildup:<first:M  partition:rest:M>;
buildup:<value pair> ≡ < cons:<value 1:pair>  2:pair>.
```

Finally, if *exprlist* is the sequence of (suspended or perhaps already evaluated) expressions, the invocation

```
partition:frons:<clock:t scramble:exprlist>
```

returns a sequence of two items which we shall call "done-pending". The first item in the pair is the list of values which converged within *t* units of time; the second item is the multiset of those which *might* not converge. The accuracy of the timing partition is somewhat dependent on the size of *exprlist* compared to *t*, the clock setting. We must take care to use sufficiently large values of *t* compared to its size in order to ensure equitable timing (just as one shouldn't clock processors with just a few nanoseconds).

The solution to the originally stated problem is then 2:done-pending which may be viewed as a sequence. If viewed as a multiset,

```
frons:<clock:t'  2:done-pending>
```

provides resumption of the unconverged computations for *t'* more units of time, just as a scheduler might resume these processes.

Implementing construction

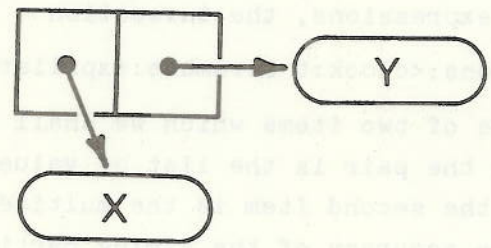
In this section we present the code for the constructors, *cons* and *frons*. The implementation is temporarily incomplete, because we say little about the implementation of the probing functions, *first* and *rest*. We can implement *null*, however, as a predicate which tests a

```
type
  pointer = ↑node;
  field = packed record
            exists : Boolean;
            value : pointer
          end;
  more = packed record
           sinker : Boolean;
           d : field
         end;
  node = packed record
           case atom : Boolean of
             true : (pname : packed array[1..6] of char);
             false : (a : field;
                    [ birthdate : smallint; ]
                      next : more)
       end
```

Figure 1. Type Declarations.

```
function cons(X,Y : pointer) : pointer;
var Q : pointer;
    NQ : node;
begin
  NQ.a := SUSPEND(X);
  NQ.d := SUSPEND(Y);
  NQ.sinker := true;
[ NQ.birthdate := TIME; ]
  NQ.atom := false;
  NEW(Q);
  sting Q with NQ;
  cons := Q
end

function frons(X,Y : pointer) : pointer;
var Q : pointer;
    NQ : node;
begin
  NQ.a := SUSPEND(X);
  NQ.d := SUSPEND(Y);
  NQ.sinker := false;
[ NQ.birthdate := TIME; ]
  NQ.atom := false;
  NEW(Q);
  sting Q with NQ;
  frons := Q
end
```



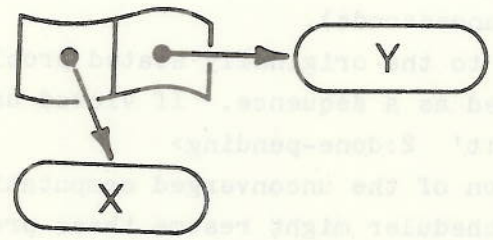


Figure 2. The Constructors.

pointer to see that it does *not* refer to a node allocated by these constructors.

We use the notation of PASCAL for the declarations and most control structures. A major exception is the **sting** operation [3] which we shall explain in two steps. For the moment we define **sting** as the only memory store operation; later it will be extended to be a conditional store operation. That is, if P is a pointer (or a memory reference) then the operation "**sting** P **with** value" is synonomous with PASCAL's "P↑ := value" assignment. Also the form "**sting** P **in** field **with** value" is synonomous with PASCAL's field assignment "P↑.field := value". All memory store operations are implemented with **sting** so that the only variable which appears to the left of the ":=" operator is a local register/variable. (As we move toward a multiprocessing environment this distinction becomes important.) The only use of the "↑" operator is, therefore, as a memory fetch. Thus, all memory operations are flagged either by "**sting**" or by "↑".

The data *type* declaration in Figure 1 prescribes the sort of node which will represent sequences and multisets. A node is either an *atom* or it is *constructed*: constructed nodes are either *sinkers* (allocated by *cons*) or *floaters* (allocated by *frons*). Floaters may eventually be *promoted* to be sinkers when necessary and when certain convergence properties are met, but sinkers never change. Sinkers are represented by rectangles in the figures; floaters are drawn as ripples (i.e. wavy rectangles).

A node has a *sinker* bit which characterizes its "shape". It also has two pointer fields, the *a-field* and the *d-field* (corresponding to *car* and *cdr*) which may refer to *exist*ent values or to *suspensions* [1]. An *exists* bit on *each* field determines whether it refers to a value or to a suspension. (In the original definition of a suspension we described it as a computation which would be coerced through complete evaluation only when necessary; here we envision that such an evaluation be fragmented into steps--each a non-trivial but finite advance from the last toward the *exist*ent value.) A node also contains a *birthdate* field which is necessary only to the fair implementation presented in the appendix.

The constructor functions in Figure 2 each allocates a fresh node, using PASCAL's *NEW* primitive, and fills all its fields appropriately. Since both the a-field and the d-field are initially filled with suspensions of the two parameters, X and Y, both these functions necessarily converge upon the successful allocation by NEW. (The fact that there is but one memory **sting** in order to fill this node is of interest in

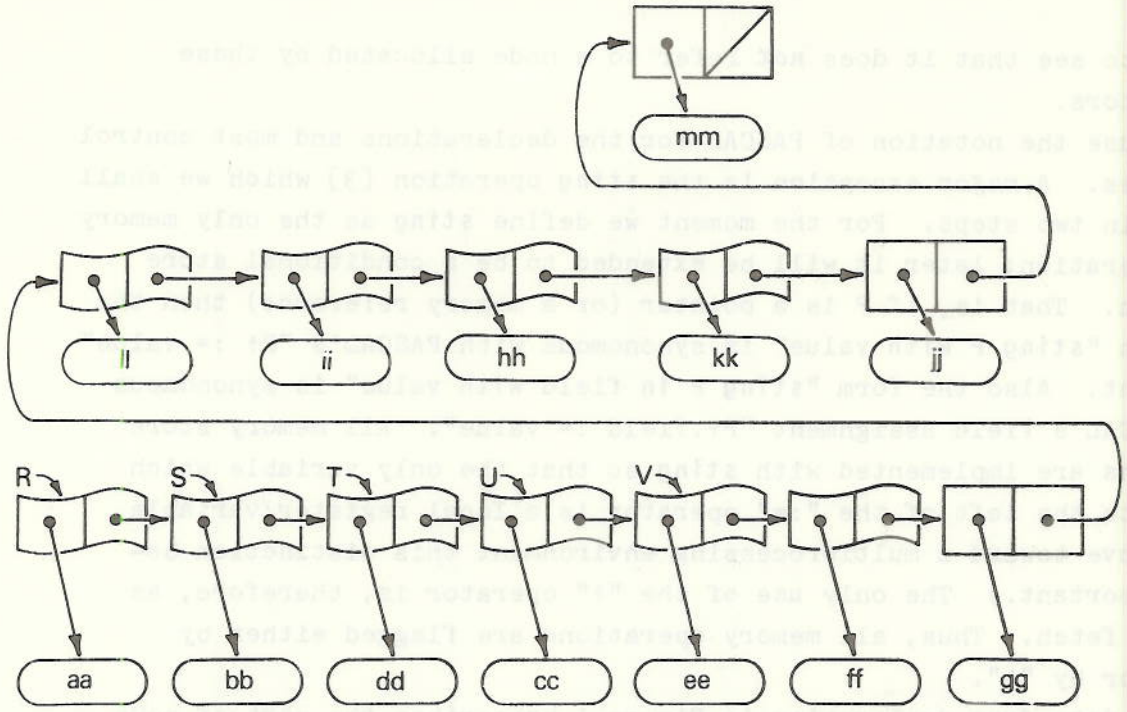


Figure 3. A fern structure referenced by R, with shared references by S, T, U, and V.

```
function first(F : pointer) : pointer;
var NF : node;
begin
  NF := EUREKA(F);
  if NF.a.exists
    then first := NF.a.value
    else NF.a := COERCE(NF.a);
         sting F [ unless a.exists ] in a with NF.a;
         first := F↑.a.value
  fi
end

function rest(F : pointer) : pointer;
var NF : node;
begin
  NF := EUREKA(F);
  if NF.d.exists
    then rest := NF.d.value
    else NF.d := COERCE(NF.d);
         sting F [ unless d.exists ] in d with NF.d;
         rest := F↑.d.value
  fi
end
```

Figure 4. Probing Functions for a Uniprocessor [or Multiprocessor].

multiprocessor environments later where we want to reduce inter-processor contention by minimizing memory manipulation. We do not, however, anticipate the requirements for the storage manager--the demands of NEW--in that environment.) The only difference between the functions in Figure 2 is the setting of the *sinker* bit.

Figure 3 illustrates the structure which might result from the following construction. Ovals denote suspensions of the appropriate values.

```
V = frons:<ee frons:<ff cons:<gg frons:<ll frons:<ii
      frons:<hh frons:<kk cons:<jj frons:<mm NIL>>>> >>>>>;
U = frons:<cc V>;
T = frons:<dd U>;
S = frons:<bb T>;
R = frons:<aa S>.
```

This figure coincides with that in another paper [2], although more shared references (denoted by single-character upper case letters) have been included in order to demonstrate that the semantics of such references will be preserved as the definitions above require.

Observation 1: The functions *cons* and *frons* always return a value, a pointer to an unshared node.

Observation 2: The function *cons* allocates a sinker, and the function *frons* allocates a floater.

The definitions of the (uniprocessor) user functions *first* and *rest* in Figure 4 [without bracketed code] are straightforward except for the function EUREKA. EUREKA is a system function upon which *all* the semantic problems fall and--in the case of concurrent processors--which will bear the responsibility for synchronization; the remainder of this paper is devoted to it. Briefly, EUREKA will search all floaters up to and including the first sinker (if there is one) accessible through successive d-fields. If there is more than one such node, it finds one which has a convergent a-field and then alters the data structure (consistently with Equations 3, 4, 6 and 7) so that the node at the front of the structure *is* a sinker; its value is the content of that node. EUREKA amounts to a simple memory fetch if that node is already a sinker. After EUREKA has returned the contents of the first node, it only remains to *coerce* any suspension which might remain in the a-field (d-field) which *first* (*rest*) must return. That coerced value must be stung into the node in memory so that further probes of that field will find the exact same value.

Observation 3: *First* and *rest* return values stored in memory.

Observation 4: (Uniprocessor) In any sinker (node) at most one

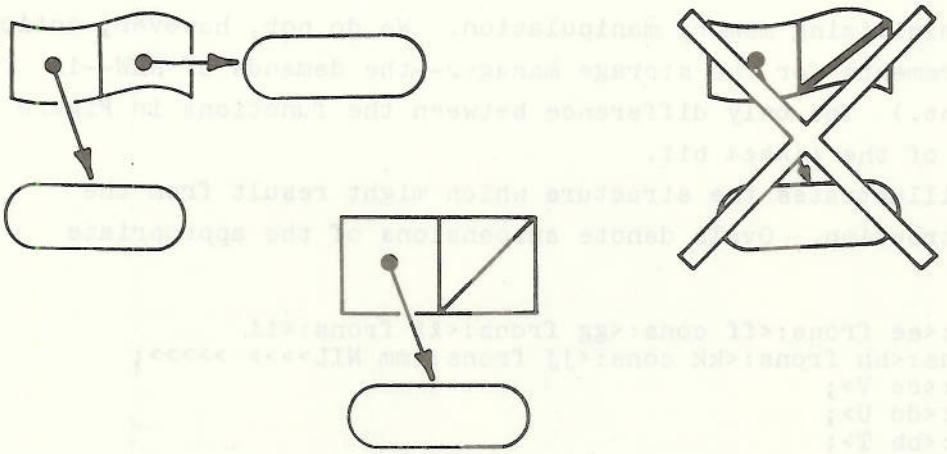


Figure 5. Floaters never have null *d*-fields.

```
function COERCE(AFIELD : field) : field;
begin
  repeat
     AFIELD := COAX(AFIELD);
   until AFIELD.exists
  taeper;
  COERCE := AFIELD
end

function COAXA(AFIELD : field, Q : pointer) : Boolean;
begin
  if not AFIELD.exists
    then AFIELD := COAX(AFIELD);
         sting Q [ unless a.exists ] in a with AFIELD
  fi;
  COAXA := AFIELD.exists
end

procedure COAXD(NEXTFIELD : more, Q : pointer);
begin
  NEXTFIELD.d := COAX(NEXTFIELD.d);
  if NEXTFIELD.value = nil
    then NEXTFIELD.sinker := true;
         sting Q in next with NEXTFIELD
    else sting Q [ unless d.exists ] in d with NEXTFIELD.d
  fi
end
```

Figure 6. Evaluation by COAXing on a Uniprocessor [or Multiprocessor].

existent value ever occupies the d-field (respectively, a-field).

Discussion: After an *existent* value is stung into a sinker all other probes will find it and won't try to change it. The only potential changes to *any existent* field occurs in the d-fields during promotion (see EUREKA below).

Observation 5: A floater never has a null d-field.

Discussion: Whenever a *nil* value is stung, the node type is stung with it (Figure 5).

The system evaluation functions COERCE, COAXA, and COAXD for a single processor are all displayed in Figure 6 [without bracketed code]. They are all built upon the assumed elementary evaluation step, COAX, which is defined according to whatever language semantics is desired, including, of course, those presented here [2]. COAX is a function which takes a suspension as an argument and returns a field as a value; that field may have its *exists* bit *true* and its pointer referring to an *existent* value, or it may have its *exists* bit *false* and its pointer referring to another suspension. Such a new suspension must represent a proper advance in the computation (or process) represented by the initial suspension; the exact amount may be presumed to be random yet finite. It is sufficient that COAX iterate a few times through the innermost loop of the evaluator and then put the (suspended) process back to sleep. COERCE is nothing more than a repeated COAXing until the *existent* value appears. The code uses a generalized form of the *repeat* loop which allows for two distinguishable kinds of exits. (D.S. Wise, D.P. Friedman, S.C. Shapiro, and M. Wand. Boolean valued loops. *BIT 15*, 4 (December, 1975), 431-451.)

Postulate: COAX makes non-trivial progress in advancing a suspended computation.

Observation 6: COAXA and COAXD properly advance the suspended computation known to some processor.

Discussion: (Uniprocessor) Each fetches a suspension, coaxes it, and restores the result if an *existent* value is not present. Progress depends on the postulate.

COAXA is declared as a function whose Boolean value tells whether its one application of COAX was the one which yielded an *existent* value. In any event, the coaxed field must be delivered to memory at Q. Of course, if the value already *existed* there, COAXA trivially returns *true*. COAXD, similar in effect but different in form is defined as a procedure since we never need to know if it uncovers a value, and it is only invoked when a suspension is known to be in the d-field so it need

not test the *exists* bit. In the event, however, that the result of coaxing in the d-field is the *existent* value *nil*, then COAXD must **sting** this value in such a way that the stung node becomes a sinker. Observation 5 is sustained in COAXD (as in all code which changes the d-field of a floater) because there is again a special test (after the value to be stung is available) which determines the way it is to be stung. If that value happens to be *nil*, then the field stung will include the *sinker* bit so that the node must become a sinker as the *nil* is stung. Observation 5 motivated the declaration of the d-field in Figure 1 to be associated with the *sinker* bit in the *more*-field. This declaration provides the more-field which is the target of the **sting** when the d-field is receiving the value *nil*.

Uniprocessor EUREKA

All implementation problems are now placed upon EUREKA. We present the code for the uniprocessor EUREKA in Figure 7 and then argue for its correctness. The reader is referred to the examples in Figure 3 and Figure 8 as this code is introduced; Figure 8 illustrates EUREKA's effect upon Figure 3 when the suspension cc is found to converge to CC. The appendix completes a full 83 line version of EUREKA which we claim is a "fair" implementation for arbitrarily many processors. We cannot justify that claim in this paper, which is only an approach to that argument, but we do use the line numbering convention from the full blown version here and in the following sections. Here we present a multiprocessor and a uniprocessor version of that EUREKA operator which are each more severe abbreviations of the code in the appendix. We develop the final version by introducing a simple version and then by adding more lines in the later one; with a few exceptions (denoted by daggers), once a line occurs in EUREKA, it is identical in all following versions. Once the uniprocessor version is verified we need only show that the additional lines do not destroy its validity under the new operating requirements.

All the local variables denote register-variables in a sophisticated system. Associated with the pointers Q and W are the node registers, NQ and NW which reflect.their recent content. Memory operations, as mentioned earlier, are explicitly denoted by **sting**. EUREKA is divided into two phases: Lines 6 through 32 are the *coaxing strategy* and Lines 33 through 82 are the *promotion*. The coaxing strategy must traverse all floaters up-to-and-including the first sinker accessible through successive d-fields, and COAX all a-fields so encountered. The loop in Lines 16 through 26 performs this function on a single processor. Whenever

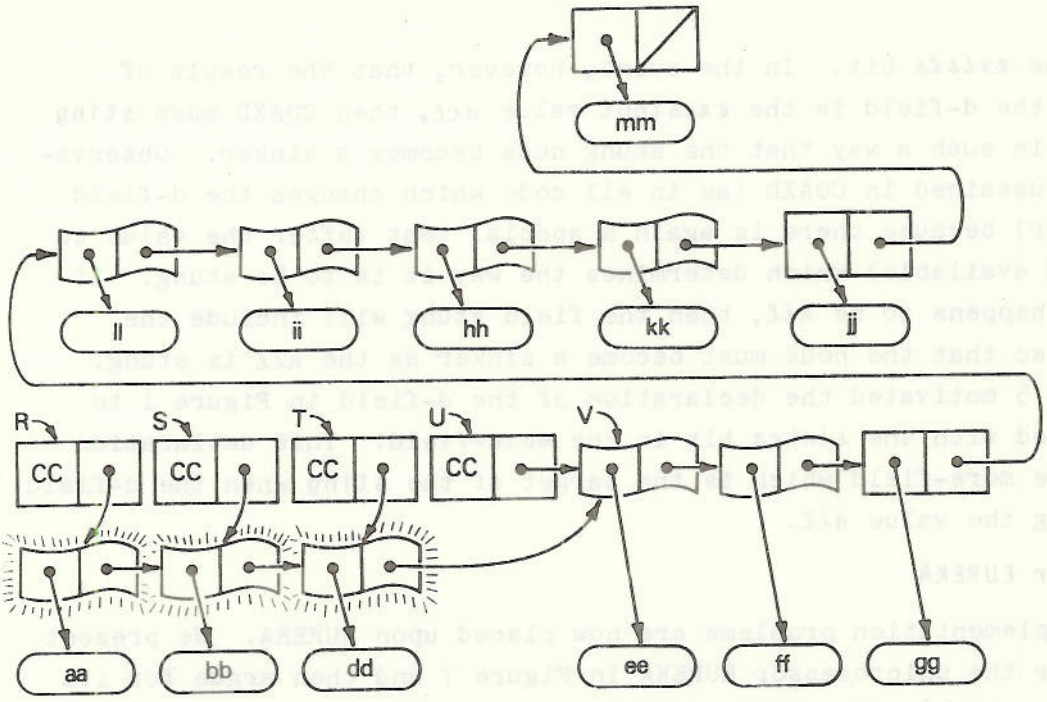


Figure 8. *first*:R = CC; shared references, as well, see CC as their *first*. If there were no shared references then no new nodes need be allocated.

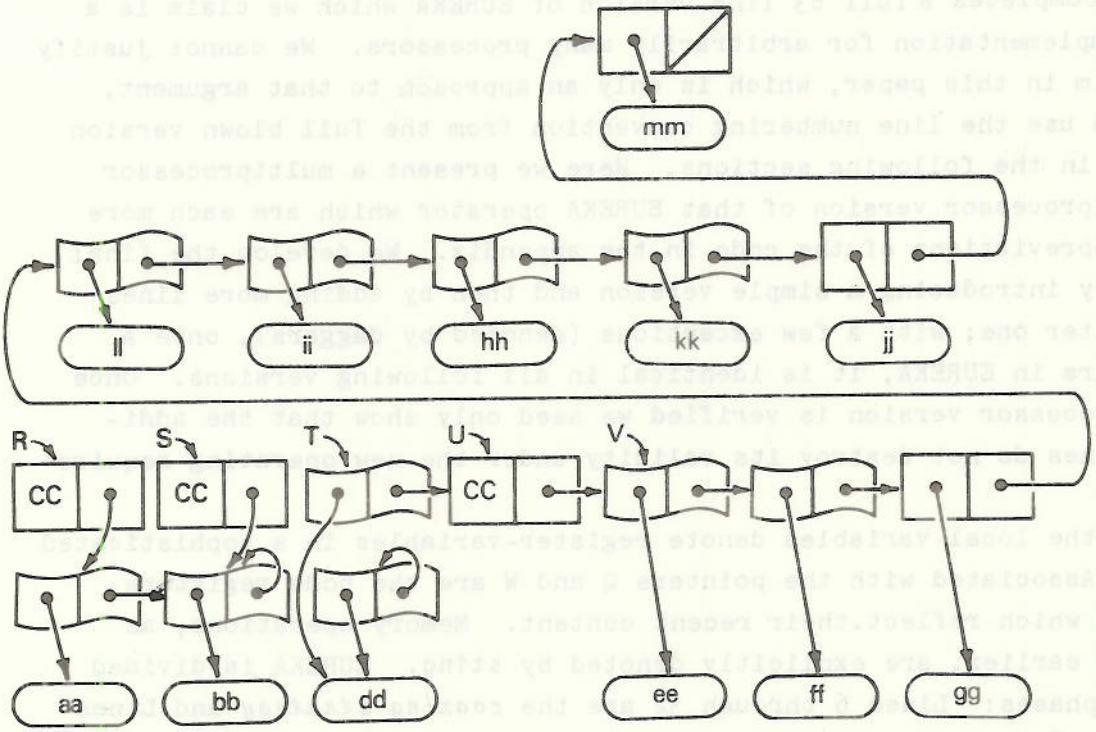


Figure 10. Reflexive pointers *during* promotion on a multiprocessor.

any a-field is *existent* then the coaxing strategy may cease. There are several ways that one coaxing pass may terminate; some terminate the entire coaxing strategy and some merely cause another pass over the data structure from the top, referred to by the pointer F. If F, itself, refers to a sinker, then promotion may be skipped and its contents may be returned as the value of EUREKA. If the accessible d-fields all refer to floaters, but the last which refers to a suspension, then that d-field suspension must be coaxed along with all the floaters' a-fields because a convergent a-field may occur in a floater not yet accessible. (Thus unbounded multisets are allowed although only an ever growing finite prefix will be coaxed. Because of the convention on null d-fields, we need only detect a suspended d-field in a floater or a sinker in order to terminate a coax pass).

The fetch at Line 35 is somewhat redundant here since COAXA has the contents of Q at Line 25. As in *first* and *rest* which refetch an *existent* value just stung, we tolerate this inefficiency in order to allow easy generalization to the multiprocessor version.

At Line 39 the pointer W refers to a node which has an *existent* a-field and NW is a copy of its content. That value will be moved to the head of all structures accessible by references to nodes between F and W (along the chain of d-fields). The necessary transformation requires the introduction of new nodes to hold intervening a-field suspensions and the planting of NQ.a.value in all the intervening nodes as shown in Figure 8. In Lines 49 through 55 NW is set up as a prototype node to be stung at R, S, and T in Figure 8 except for different d-fields. (We know that all d-fields between F and W do *exist*.) Promotion is performed in the loop from Line 64 to 73, part of which is duplicated as Lines 56 to 60 treating the node at F as a special first case. After that first node, the new contents of F--*the value of the EUREKA function*--is determined and F is used as a traversal pointer thereafter.
Lines 74 through 79 fill in the d-field of the last node introduced by promotion, and therefore require the special test for null d-fields. SUFFIX is essentially the original d-field of the promoted node NW; if that is suspended then at Line 79 we are very careful not to copy that suspension-- rather we create an indirect pointer to the extant one so that it will eventually only be evaluated once!

Observation 7: There are no (non-trivial) circular paths in the system.

Observation 8: The coaxing strategy will find a convergent candidate for promotion if one exists.

Discussion: It coaxes on all suspensions up through the first

sinker uniformly.

Observation 9: Promotion makes the *first* node in a structure into a sinker containing an *existent* value.

Observation 10: Promotion preserves the structure semantics.

Discussion: The proof is by induction on the distance from the node pointed to by F and the node pointed to by W. Each sharing causes a new node to be created as the d-value of a sinker.

Theorem 1: The uniprocessor implementation is correct.

Multiprocessor EUREKA

The modification of EUREKA to run on arbitrarily many processors is the centerpiece of this paper (Figure 9). The coaxing strategy proceeds on many processors *without* synchronization of the *processors*. Promotion may proceed simultaneously with coaxing, but only one promotion may be active at a time. Thus, we introduce a binary semaphore, MUTEX, at Lines 33, 52, and 81 to protect that part of the program which distorts the data structure in order to effect promotion.

The fact that coaxing, which is the more expensive computation, may proceed without processor synchronization is due to the **sting** primitive which we now extend to its conditional form. We introduce this primitive separately in another paper [3], but this example is a better demonstration of its power. The operation "**sting** P **unless** bit **in** field **with** value" dispatches a description of a bit location and a field location within a memory word, and a value to the location at P. There (not *in* the dispatching processor) an uninterruptible test-and-store operation proceeds; if the *bit* in that word is already *true* then nothing happens; if the *bit* is *false* then the *value* is stored in the *field*. The dispatching processor receives no feedback on what actually happens!! Because there is no feedback the dispatching processor is free to proceed immediately. Because there is no waiting for feedback (as there is with a *test-and-set*) no processor may monopolize a path to memory during an uninterruptible memory operation, and there will be fewer instances of one processor blocking another's access to memory during the storing of sensitive data. Viewed another way, the **sting-unless** primitive avoids implementing every *exists* bit as a semaphore.

The major problem with multiprocessor coaxing under EUREKA is that we do not allow a second *existent* value to be stored over an earlier one in any field. This might occur, for instance, if two processors picked up the same suspension at about the same time, coaxed it to an *existent* value, and then stored two different incarnations of that value back at two separate times. (If that *existent* value were a floater then Equa-

tions (6) and (7) might be violated through duplication of what ought to be a single floater.) Thus in Figures 4 and 6 whenever a **sting** is directed *in* an a-field (d-field), the sting must be conditional--**unless** a.exists (d.exists) as indicated by the bracketed code.

Coaxing may run on many processors at once, but not without a benign sort of interference called *regression*. If several processors are coaxing the same suspension, then all might make the same progress. If one of those processors were inordinately slow, then the remainder might have made considerable coaxing progress on that same field--far beyond just one coax, but not yet *exis*tent-- before the sloth processor delivers its duplicate (now stale) effort. Since none of these processors have stung an *exis*tent value, anyone may yet sting this field. When the sloth processor delivers its suspension, all future coaxes will begin from there--even those coaxes by the speedy processors which had already coaxed the value much further along. These processors may appear to be dragged backwards, but this backward progress or regression is limited by the number of active processors in the system. In the worst case, all processors are coaxing at one field, and the time to coerce will be no worse than the slowest processor. Thus the coaxing strategy in Figure 9 is sufficient for one, two, or two hundred simultaneous incarnations of EUREKA.

We modify the discussion of a previous observation:

Observations 6's discussion (Multiprocessor): In this case the COAXing advances the suspension known to the coaxing processor. If stinging the result is *un*successful, then there is a value in that field, so no further advancement is necessary.

Promotion is a different story because it alters the *exis*tent d-fields. Only one promotion may proceed at a time; promotion is a critical region protected by the binary semaphore MUTEX, but it is a very fast transformation because it involves no coaxing and only two traversals of the structure from F to W. (**Sting-unless** is also sufficient to implement a binary semaphore if we do not require fairness from it [3].) Because of the probable wait at Line 33 when the structure might change, the value to be promoted must be relocated by a pre-pass through the structure at Lines 34-38. If a change did occur as a result of another promotion during the wait, it can only simplify the waiting promotion.

Observation 4: (Multiprocessor) In any sinker at most one *exis*tent value ever occupies either field.

Discussion: The only change from the uniprocessor version of this invariant is that the a-fields of floaters may change. That might occur

MULTIPROCESSOR

```
 1.    function EUREKA(F : pointer) : node;
 2.    var Q, W, FNEXT : pointer;
 3.        SUFFIX : more;
 4.        NQ, NW : node;
 5.    begin
 6.      NQ := F↑;
 8.      Q := F;
16.      repeat
17.          if NQ.sinker
18.            then if Q = F then return(EUREKA := NQ) else Q := F fi
19.            else if NQ.d.exists
20.                   then Q := NQ.d.value
21.                   else COAXD(NQ.next,Q); Q := F
22.                 fi
23.          fi;
24.          NQ := Q↑;
25.        until COAXA(NQ.a,Q)
26.      taeper;
*29.     Q := F;
*33.     P(MUTEX);
*34.     repeat
35.          NQ := Q↑;
*36.       until NQ.a.exists;
*37.       Q := NQ.d.value
*38.     taeper;
39.      W := Q; NW := NQ;
49.      NW.sinker := true; sting W in sinker with true;
51.      if W = F then
*52.                V(MUTEX);
53.                return(EUREKA := NW)
54.      fi;
55.      SUFFIX := NW.next; NW.d.exists := true;
56.      NQ := F↑; FNEXT := NQ.d.value;
57.      NEW(Q);
*58.     NQ.d.value := Q;
59.      sting Q with NQ;
60.      NW.d.value := Q; sting F with NW;
62.      EUREKA := NW;
64.      repeat
65.        until FNEXT = W;
66.          F := FNEXT;
67.          NQ := F↑; FNEXT := NQ.d.value;
68.          NEW(Q);
*69.         NQ.d.value := Q;
70.          sting Q with NQ;
71.†         sting NW.d.value in d with NQ.d;
72.          NW.d.value := Q; sting F with NW
73.      taeper;
74.      if SUFFIX.exists
75.        then if SUFFIX.value = nil
76.               then sting Q in next with SUFFIX
77.               else sting Q in d with SUFFIX.d
78.             fi
79.        else sting Q in d with SUSPEND("rest(W)")
80.      fi;
*81.     V(MUTEX)
83.   end
```

Figure 9. Multiprocessor EUREKA.

must sting unsuccessfully.

Observation 13: Only one promotion proceeds at a time.

Observation 14: Promotion does not interfere with coaxing.

Discussion: If coaxing occurred at the promotion site, either it would find a promoted value and quit, or proceed (perhaps circularly) oblivious to the promotion thereabouts.

Observation 15: Lockup is impossible.

Discussion: EUREKA cannot be invoked directly or indirectly during promotion. Once promotion begins, it will find an *existent* value and eventually execute the V operation (Lines 52 or 81) terminating promotion.

Theorem 2: The multiprocessor version is correct.

Conclusion

We have briefly introduced a new constructor for multiprogramming, which allows a facile expression of complicated order interdependencies with a familiar programming style. The proof techniques for such programs, although not treated directly here, promise to be as straightforward as one can hope for since the language has an applicative style which yields nicely to inductive proofs.

The emphasis in this paper has been on the uniprocessor and multiprocessor implementations of the semantics for this language. The problems of efficient implementation have motivated new control structures, notably the **sting-unless** (Others are suggested in the appendix) which has simplified the interprocessor protocol. We have argued informally that the implementations of the constructors, *cons* and *frons*, are correct for this semantics, but these implementations are not the ultimate goal of this work.

In order to establish *frons* as a constructor truly suitable for applicative multiprogramming we must address the problem of fairness proofs. For nondeterminism, such fairness might imply that any convergent value in a multiset could not be ignored forever. (One might make a more precise statement, but this will do.) A scheduler may be correct without being fair if it loses a job it's supposed to be timing, because the output from the other processes might be correct. That doesn't mean that such an operating system is useful, however; good systems don't lose processes.

The uniprocessor system is implemented, and a multiprocessor version can be simulated on a single processor. A good implementation of the fair version is an exciting prospect for machines with a high degree of parallelism (e.g. a hundred processors).

Acknowledgement

We thank several colleagues who helped in the development of EUREKA, especially by sitting still while the problem *and* this solution were described. Particular thanks are due to Mitchell Wand who raised important questions that redirected early attempts to "find it." The research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS75-06678 A01 and MCS77-22325.

References

1. D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, S. Michaelson and R. Milner (eds.), Edinburgh, Edinburgh University Press (1976), 257-284.
2. D.P. Friedman and D.S. Wise. Applicative multiprogramming. Technical Rept. No. 72, Computer Science Dept., Indiana University (December, 1978).
3. D.P. Friedman and D.S. Wise. A conditional, interlock-free store instruction. Preliminary version in M.P. Pursley and J.B. Cruz, Jr. (eds.), *Proc. 16th Allerton Conf. on Communication, Control, and Computing*, Univ. of Ill., Urbana (1978), 578-584.
4. P. Henderson and J.H. Morris, Jr. A lazy evaluator. *Proc. 3rd ACM Symp. on Principles of Programming Languages* (1976), 95-103.
5. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 7 (July, 1978), 558-565.
6. J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (eds.), Amsterdam, North-Holland (1963), 33-70.
7. Z. Manna. *Mathematical Theory of Computation*, New York, McGraw-Hill (1974), 418.
8. J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. *J. Comp. Sys. Sci. 9*, 3 (June, 1974), 332-354.
9. C. Wadsworth. *Semantics and Pragmatics of Lambda-calculus*, Ph.D. dissertation, Oxford (1971).

Appendix

We present a complete version of EUREKA which represents the unattained goal of this paper. We intend that it run in the same processor-rich environment as the multiprocessor version, but that it be "fair". Fairness includes the guarantee that no accessible suspension ever go uncoaxed; the procedure COAXSUFFIX serves this purpose:

```
procedure COAXSUFFIX(NQ : node, Q : pointer);
begin
  if not repeat
           until NQ.sinker;
           while NQ.d.exists;
             Q := NQ.d.value; NQ := Q↑;
             COAXA(NQ.a,Q)
         taeper
    then COAXD(NQ.next,Q)
  fi
end
```

Moreover, we must guarantee that no converged value ever go unpromoted. The solution is a system clocking mechanism like Lamport's [5] used to referee the choice of which value gets promoted. The time when a node is created is installed in that node by *cons* or *frons* and it remains with that node and its copies forever; given a choice, we always promote the "oldest" value. (Rather than providing one field in every node for an unbounded birthdate, Guy Steele, Jr., has suggested that a pointer field would suffice for linking the finite number of accessible nodes together in order of their creation. The storage requirements would then be quite tractable, but the determination of the relative age of two arbitrary nodes at Line 44 becomes a computation whose time is linear in the number of nodes; here that determination takes only constant time.)

The code below only includes lines which are new to or changed in the fair version of EUREKA. As before a dagger (†) denotes a line slightly changed by additions. The new lines which assure fairness are noted by asterisks (*). Several new lines are included to assure efficiency, particularly to avoid the critical region (promotion) as much as possible; these are noted by dollar signs ($). The process creation suggested by **spawn** at Lines 15 and 61 may be ignored; these lines may be read here simply as procedure invocations and Line 82 (related to Line 61) may also be ignored.

FAIR

```
$ 7.    if NQ.sinker then return(EUREKA := NQ) fi;
* 9.    if repeat
*10.         until COAXA(NQ.a,Q);
*11.         while not NQ.sinker;
*12.         while NQ.d.exists;
*13.           Q := NQ.d.value; NQ := Q↑
*14.       taeper
*15.     then spawncall(COAXSUFFIX(NQ,Q))
 16.†    else repeat
*27.    fi;
$28.    if Q = F then sting F unless d.exists in sinker with true
 29.†     else Q := F
$30.    fi;
$31.    NQ := Q↑;
$32.    if NQ.sinker then return(EUREKA := NQ) fi;
*40.    repeat
*41.      until NQ.sinker;
*42.      while NQ.d.exists;
*43.        Q := NQ.d.value; NQ := Q↑;
*44.        if NQ.a.exists andif NQ.birthdate < NW.birthdate
*45.          then W := Q; NW := NQ
*46.        fi
*47.    taeper;
$48.    if not NW.sinker
 49.†     then NW.sinker := true; sting W in sinker with true
$50.    fi;
$61.    spawnreturn(
$63.                );
$82.    quit
```