WORKING PAPER 148

May 1977

A History Keeping Debugging System for PLASMA

Jerry Morrison

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Working papers are informal papers intended for internal use. This report describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided by the Office of Naval Research under contract N00014-75-C-0522.

A History-Keeping Debugging System For Plasma

by

Jerry Howard Morrison

Submitted to the Department of Electrical Engineering and Computer Science on May 27, 1977 in partial fulfillment of the requirements for the Degree of Bachelor of Science.

ABSTRACT

PLASMA (for PLAnner-like System Modeled on Actors) is a message-passing computer language based on actor semantics. Since every event in the system is the receipt of a message actor by a target actor, a complete history of a computation can be kept by recording these events. The facility to search through and examine such a history, combined with the facility to pre-set breakpoints or stopping points, and the ability to restore side effects, provides a powerful way to debug programs written in PLAMSA. The kinds of history-manipulation and breakpoint setting commands needed, and the ways they can be used, particularly on recursive programs without side effects, are presented.

Dr. Carl E. Hewitt, Thesis Supervisor

Associate Professor of Computer Science and Engineering

Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology

DEDICATION

This thesis is dedicated to the Institute, which has the finest professors.

TABLE OF CONTENTS

BACKGROUND

PLASMA is a message-passing computer language based on actor semantics. For completeness, a partial description of PLASMA syntax and semantics is given below, enough to understand all the examples in this thesis.

PLASMA is an applicative language, like LISP, but has several basic differences. All program elements in PLASMA are actors. An actor is defined to "do something" when it receives a message, which is also an actor. Each and every event in a computation is the receipt of a message.

Every actor "knows about" certain other actors, its acquaintances. In terms of tree-machine implementations of LISP, these correspond to the *needed* variable bindings in closures. An ENVIRONMENT is a set of identifier bindings, extendable by pattern matching (explained below) and by the LET function (not explained here).

The basic data structure in PLASMA is the SEQUENCE, which is a generalization of the list, except they are pure. An example of a sequence is:

```
[10 20 [30] 40]
```

When a sequence is evaluated (i.e., sent an EVALUATE message) the result is a new sequence, containing the results of evaluating each of the elements. Since numbers evaluate to themselves, evaluating the above sequence would produce an equivalent one. An UNPACK evaluated within a sequence (UNPACK is a prefix symbol which looks like !), will "open up" an inner sequence. For example, if s is bound to the above sequence, evaluating:

```
[1 2 !s 3]
```

would yield:

```
[1 2 10 20 [30] 40 3]
```

PLASMA syntax is a bit more readable than LISP. One place this difference appears is in the evaluation of FORMS. A form is a structured object much like a SEQUENCE, except it is written with parentheses. Some example forms:

```
(+ 100 1)
(f x)
(f (f (f x)))
(message => target)
```

Since forms are used for functional application, they evaluate in an analagous fashion to LISP, e.g., prefix notation is generally the rule. However, there are five special symbols which, if one of them is the second element of some form, determine how to evaluate that form. These are: =>, <=, ->, <-, and ==. The semantics of these infix symbols will be explained later.

There are several prefix symbols in PLASMA (like QUOTE in LISP), some of which will be explained later: quote ', quasi-quote ", unpack !, binder =, and variable-binder @=.

Receivers are actors that employ pattern-matching to bind their incoming messages to internal identifiers. If the pattern does not match some incoming message, the receipt of that message causes a NOT-APPLICABLE error. A receiver looks like this:

(=> *pattern body*)

where *pattern* is the pattern to match against each incoming message, and *body* is one or more expressions to evaluate in the new environment. The new environment is the

environment the receiver is defined in, extended by the matching of *pattern* against the incoming message.

The common patterns are:

?, which matches any object actor.

A quoted identifier, which matches the same identifier.

An identifier, which matches an object if the identifier's value is equivalent to that object.

=identifier, which matches any object, and binds identifier to that object.

A sequence, which matches an object that is also a sequence, if the elements of the pattern sequence match those of the object. Inside a sequence pattern, the pattern !=identifier will match against any number of elements from the object (including zero), and bind identifier to a sequence of those elements. Also, !? will match against any number of elements from the object, with no binding.

@=identifier, which matches any object actor and binds identifier as a variable, initially set to that object. This is explained below.

Messages are commonly sequences, so the receiver:

(=> [=x_1 =x_2 ... =x_n] *body*)

is analagous to the LISP form:

(LAMBDA (x_1 x_2 ... x_n) *body*)

since, if the incoming message is a sequence of n elements, it will bind the identifiers x_1

through x_n to those elements.

CASES implements conditionals using pattern matching. A CASES contains several receivers; when CASES receives a message, it tries to send it to each of them in turn. The first receiver that will accept the message gets it. An example of CASES is:

```
(CASES (=> [] body1)
       (=> [=first !=rest] body2)
       (else body3))
```

This simply evaluates $body_1$ if the incoming message is an empty sequence, evaluates $body_2$ if it is a non-empty sequence, and evaluates $body_3$ otherwise. Note that $body_2$, if evaluated, will be evaluated in an environment where the identifier `first` is bound to the first element of this sequence, and the identifier `rest` is bound to the rest of this sequence (i.e., a sequence of the rest of the elements). In the third receiver, "`else`" is just a nice abbreviation for "`=> ?`".

The message `m` can be **TRANSMITTED** to the target actor `t` simply by evaluating:

```
(m => t)
```

or, equivalently:

```
(t <= m)
```

Alternatively, functional application:

```
(function arg1 arg2 ... argn)
```

means transmit:

```
(function <= [arg1 arg2 ... argn])
```

If the second element of a form is <u>not</u> one of the special symbols `=>`, `<=`, `->`, `<-`, or `==`, then evaluating that form means "apply the first element as a function", i.e., send a sequence of

the rest of the form's elements to the first element. Note that a function does not have to have its "arguments" evaluated (receiver (=> *pattern body*) is a case in point), although the mechanism for defining such a function is not described here.

We can define a function in two basic ways, illustrated here by defining ADD-3 and MULTIPLY:

```
(ADD-3 <- (=> [=x] (+ x 3)))
```

and:

```
(define (MULTIPLY =x =y) (* x y))
```

which is equivalent to:

```
(MULTIPLY <- (=> [=x =y] (* x y)))
```

The arrow <- simply does an assignment in the top-level environment here. Assignments in general will be described shortly. Now we can use them:

```
(ADD-3 5)    yields   8
(ADD-3 <= [10])    yields   13
(MULTIPLY 3 6)    yields   18
([4 7] => MULTIPLY)    yields   28
```

Primitive side effects (assignments) are implemented with **variables**. Assignments can be done on all identifiers in the top level environment, and on identifiers bound as variables in local environments. One way to make an identifier a variable in some environment is by binding it using the pattern @=*identifier*. If an identifier (say, X) is *hard bound* to some object (by one of the patterns =X or !=X), then it is just a local name for that

object, and cannot be altered. If an identifier is bound *as a variable* to some object, then it refers to a cell whose contents can be changed.

Evaluating an identifier gets the object it is bound to, whether it is hard bound or variably bound. In the top level environment, all variables can be variably bound. We can politely ask a variable (say, C) to change its value to that of *expression* by evaluating:

(C <- *expression*)

or, equivalently:

(*expression* -> C)

which returns the value of *expression*. Note that <- doesn't evaluate its first argument (to the left of the arrow), and -> doesn't evaluate its second argument (to the right of the arrow).

It is often necessary to refer to an entire actor within itself (recursive reference). Functions defined at top level can do this easily since the assignment of a definition to an identifier in the top-level environment provides a general handle on that definition. For local environments, == is used:

```
(name == expression)
```

This evaluates expression in an environment in which the identifier name is bound to the results of this very evaluation. It is an error if the value of name is actually needed before the evaluation is complete.

A SMALL PROGRAM

For an example of the use of variables and ==, we can implement an "impure" CONS (one whose *first* and *rest* elements can change) in this way:

```
(define (CONS e=first-cell e=rest-cell)
 (cons-cell ==
  (CASES (=> 'first first-cell)
         (=> 'rest rest-cell)
         (=> ['replace-first =new-first]
             (first-cell <- new-first) cons-cell)
         (=> ['replace-rest =new-rest]
             (rest-cell <- new-rest) cons-cell) )))
```

We now can CONS two things x and y together by evaluating (CONS x y); the result is a "cons-cell". We can get x (the first) back by sending this cons-cell a `'first` message. We can replace x with z as the first element by sending the cons-cell the message `['replace-first z]`; the result being the cons-cell itself, now changed. If this cons-cell is again sent the message 'first, the answer will now be z.

KEEPING A HISTORY

When a message is TRANSMITTED, an EVENT occurs, where the MESSAGE is sent to the TARGET, along with a CONTINUATION. This event is called a REQUEST. Eventually, there should be a matching REPLY event, in which this CONTINUATION is the TARGET. This is where the sender "gets its answer". There is no CONTINUATION in a REPLY event.

There are three tasks involved in keeping a history during a running computation. First, all events must be recorded. This means remembering, for each event, the MESSAGE, the TARGET, whether the event is a REQUEST or a REPLY, and the CONTINUATION (if the event is a REQUEST).

The second task is to record the primitive side effects that occur. When a variable is asked to change its value, the history-keeper must make special note of this, and remember the variable and its old value. With this information, the old value can be restored when looking back through the history. This also means that part or all of a computation can be *undone*.

The third task of the history-keeper is to monitor the events as they occur, and match each against the user-specified patterns for breakpoints (if any). When such an event is about to occur, the history-keeper causes a BREAK, which starts the debugger's READ-EVAL-PRINT loop (see the next section). The possible specifications for such break conditions could be arbitrarily complex; however, since these conditions must be tested for each and every event, overly-complicated break conditions will slow down the primary computation excessively.

LOOKING AT THE HISTORY

The user, debugging his programs, talks to the debugging system itself through a READ-EVAL-PRINT loop. A READ-EVAL-PRINT loop reads an object from the console, evaluates it, prints the answer, and then does it all over again. The debugger's READ-EVAL-PRINT loop responds to commands to manipulate the history also. Until the user runs into an error or asks for the debugger, it remains unseen and unheard; albeit using up some processor time and memory space recording the history.

The debugger loop can be summoned by a direct request from the terminal, by the explicit evaluation of (BREAK) in a program, by hitting a breakpoint, and by an interpretation error.

For any significant computation, the history will be quite long and detailed. One must therefore be able to extract the desired information from the history. Fundamentally, the user wants to find and display events, environments, and actor acquaintances. A special case of this is scanning a chain of continuations. Since one of the acquaintances of a continuation actor is the previous continuation, they form a chain of computational steps.

There is a pointer associated with history observation that indicates the "current" event. The EVENT DISPLAY functions allow the user to examine the current event: display the type of event, the message, the target, the continuation, and each of their acquaintances. Other functions just retrieve pointers to these objects, so they can be used as arguments to the search functions (described below). In addition, it is very useful to be able to display function definitions, with an imbedded cursor in each active one showing at what point the current evaluation of that function is.

The basic MOVE commands move the history pointer, changing the conceptually "current" event. These include:

move forward or backward one event;

move forward to this REQUEST's matching REPLY;

move backward to this REPLY's latest or earliest matching REQUEST (note that one REPLY event can serve for several REQUEST events, which is what happens for iteration and buck-passing; see [Hewitt, December 1976]);

move forward or backward to the next call to any user-defined functions;

move forward or backward to the next top-level READ-EVAL-PRINT loop event [the normal command/expression loop's stopping points]; and

move to the beginning or the end of the history.

More powerful motion is accomplished with the SEARCH commands, which search the history forwards or backwards for events referencing certain actors in certain ways. These include searching for an event:

with a specific function as the TARGET;

with TARGET, MESSAGE, or CONTINUATION a specific actor or a specific type of actor (e.g., any number);

where a particular variable's value is changed; or

where a particular actor first appeared in a REPLY ("where it originated").

In addition to jumping around through the history looking at selected events, the user may wish to look at a structured subset of the history. The HISTORY DISPLAY

functions print a trace of messages to and from specifically named actor(s), or to and from all user-DEFINEd functions, with or without the recursed calls. In addition to the REQUESTs to these actors and their respective REPLYs, the history display functions can include a complete trace of the computations lexically scoped within these actors. These events are just those between an initial REQUEST and its respective REPLY, excepting all those between any REQUEST(s) to some actor outside the given actor and the answering REPLY(s).

A BREAKPOINT is a stopping point in a program, set by the user through the debugger. While the program is executing, the history-keeper checks each event to see if it matches any of the user-defined patterns for breakpoints. If so, it causes a BREAK; that is, it stops execution of the program and starts the debugger's READ-EVAL-PRINT loop. The simplest patterns for breakpoints look for events with specifically named actors or functions as the TARGET. Also available during execution are commands to stop at and display the next event and to stop at and display a REQUEST's matching REPLY.

Two other commands to the debugger's READ-EVAL-PRINT loop allow the suspended computation to be continued or discontinued.

AN EXAMPLE

For an example of the history in action, we have below a simple substitution function. It will substitute a new item x for every occurence of an item y, at every level of depth within some sequence z.

```
(define (substitute =x =y =z)
 (z =>
  (CASES (=> y x)
         (=> [=first !=rest]
             [(substitute x y first) !(substitute x y rest)])
         (else z)) ))
```

This routine is fairly straightforward. If z is equivalent to y, then the answer is x. Otherwise, if z is a non-empty sequence, then the answer is a new sequence, obtained by recursively calling substitute on both the first and the rest of the sequence z. Otherwise, the answer is just z itself. Note that the empty sequence [] is included in this last case.

We might use substitute like this:

```
(substitute 0 100 [0 [100]])
```

In other words, substitute 0 for 100 in the given sequence. This will return:

```
[0 [0]]
```

Or like this:

```
(substitute 'hello 'hi '[hi there])
```

which returns:

```
[hello there]
```

or:

```
(substitute 0 'zero '[no zeros here])
```

which answers:

```
[no zeros here]
```

Now we can switch into debug mode and have a look at the history just generated by those three calls. First, search backwards for the last call to substitute. The debugger

prints:

```
REQUEST: substitute <= [0 zero [no zeros here]]
CONTINUATION: c
```

The continuation is abbreviated as c here. Now, skip forwards to the matching REPLY (where c gets its answer):

```
REPLY: c <= [no zeros here]
```

Let us look at a complete trace of calls to substitute from the first invocation. For brevity, the continuations have been named c_1, c_2, c_3 ...; numbered as they are encountered. The subscript numbers have no significance other than uniquely naming the continuations. Now, here's the trace:

```
REQUEST: substitute <= [0 100 [0 [100]]]
CONTINUATION: c1

  REQUEST: substitute <= [0 100 0]
  CONTINUATION: c2

  REPLY: c2 <= 0

  REQUEST: substitute <= [0 100 [[100]]]
  CONTINUATION: c3

    REQUEST: substitute <= [0 100 [100]]
    CONTINUATION: c4

      REQUEST: substitute <= [0 100 100]
      CONTINUATION: c5

      REPLY: c5 <= 0

      REQUEST: substitute <= [0 100 []]
      CONTINUATION: c6

      REPLY: c6 <= []

    REPLY: c4 <= [0]

    REQUEST: substitute <= [0 100 []]
    CONTINUATION: c7

    REPLY: c7 <= []

  REPLY: c3 <= [[0]]

REPLY: c1 <= [0 [0]]
```

DEBUGGING RECURSIVE PROGRAMS

The history provides us with a general paradigm for debugging recursive and iterative programs that have no side effects; the requirement is that you must be able to tell if the results are correct. It is also assumed here that the program is reasonably close to embodying a correct algorithm. Iteration control structures in PLASMA are discussed at length in [Hewitt, December 1976]. Suffice it to say here that to iterate, an actor simply sends itself a message. If the result of that message will simply be handed back to the current continuation, no new continuation is generated (as would be in recursion), but the current continuation is used again in the new REQUEST, so that it will directly receive the REPLY.

To debug a program, the first step is to load in the program definition, turn on the history-taker, and start executing the program. There are four things that can happen:

Correct Execution -- the program can execute properly and return the correct result (in which case you didn't need the debugger);

Interpretation Error -- the program can hit an interpretation error (illegal operation) and halt immediately;

Non-Termination -- the execution might never terminate (endless loop); or

Incorrect Result -- an incorrect result may be returned.

Notice that the debugger will not recognize if an answer is correct (that is, the difference between the first and last possibilities, above), and is of no help if the user cannot. Remember also that the goal of test-case debugging is to locate the bugs so they can be corrected, not to attempt to prove program correctness.

INTERPRETATION ERRORS

If the interpreter catches an error (an illegal operation such as division by zero), the program will immediately halt, and the debugger's READ-EVAL-PRINT loop will start. It is a straightforward matter from this point to scan backwards through the history from the termination of the computation to the cause of the error.

Back to our example, we can rewrite the substitute function so that it gets an interpretation error. We simply leave out the last clause in the CASES (space underlined below):

```
(define (substitute =x =y =z)
 (z =>
  (CASES (=> y x)
         (=> [=first !=rest]
             [(substitute x y first) !(substitute x y rest)])
         _____)))
```

and type:

```
(substitute 1 2 [2 2])
```

which, pretty quickly, comes back with:

```
ERROR:  NOT-APPLICABLE:
[] => (CASES (=> y x)
             (=> [=first !=rest]
                 [(substitute x y first) !(substitute x y rest)]))
```

All of which goes to say that the message [] did not match the patterns of any of the receivers in the CASES statement shown. If there is any doubt about what function this CASES appeared in, the answer could be obtained by moving backwards through the history for the last call to a user-defined function.

The problem is now obvious, a condition must be added to handle the empty

sequence in the CASES. What isn't quite as obvious is that there is no case for everything that isn't supposed to be substituted for. If the case (=> [] []) is inserted, substitute will work on the previous test, but will still fail on many others, for example (substitute 1 2 [1 2]).

NON-TERMINATION

If a computation takes a long time (a possible infinite loop), it is only necessary to interrupt the computation and search through the history for a repeating pattern of messages (a lack of convergence). If the program is converging on an answer, then simply continue the computation. If the program is diverging from an answer, the bug is either in a conditional that failed to terminate the iteration or recursion, or in the code that breaks the problem into subproblems. Otherwise, the computation is repeating itself, and the bug must be in the code that breaks the problem into subproblems.

An easy way to make the substitute function compute forever is to change the recursive invocation case. Again, the bug introduced is underlined:

```
(define (substitute =x =y =z)
 (z =>
  (CASES (=> y x)
         (=> [=first !=rest]
             [(substitute x y first) !(substitute x y z)])
         (else z)) ))
```

Let us test it on some data:

```
(substitute [1] 1 [1 0 1])
```

Now, this thing may compute for a while before we decide to interrupt it. When we do, the

debugger will display the current event, e.g.:

```
REPLY: c10 <= [1]
```

This looks reasonable enough. Back up a few REQUESTs and trace the rest of the messages sent to substitute:

```
REQUEST: substitute <= [[1] 1 [1 0 1]]
CONTINUATION: c11

  REQUEST: substitute <= [[1] 1 1]
  CONTINUATION: c12

  REPLY: c12 <= [1]

  REQUEST: substitute <= [[1] 1 [1 0 1]]
  CONTINUATION: c13

    REQUEST: substitute <= [[1] 1 1]
    CONTINUATION: c10

    REPLY: c10 <= [1]
```

It is pretty obvious, in this simple case, that the program is looping indefinitely. In more complex cases, the same ideas apply.

INCORRECT RESULT

If a computation returned an incorrect answer, we have a complete history of an erroneous computation, and we want to locate the bug(s). The basic idea is to find the deepest recursive call to the function that yields an incorrect answer, and look at all the recursive calls within that call. This is easily done by invoking a history search function to scan for calls to the routine in question, and then moving to the matching REPLY events to check the matching results. From the end of the history, searching backwards for a call to a

routine will find the deepest invocation of that routine in the last branch of the computation tree.

By the nature of recursive programs, there are four possible kinds of errors that can have been committed at this point:

1. Error in testing the recursion termination condition(s). If there are no recursed calls where there should be, or if there are recursed calls where there shouldn't be, the bug lies in testing the termination condition(s).
2. Error in handling a termination condition. If in this call the routine (correctly) did not call itself recursively, then the bug lies in the code that handles a recursion termination condition.
3. Error in decomposing the problem. If the next level down of recursed calls are made incorrectly (i.e., given the wrong arguments), the bug is in decomposing the problem into subproblems.
4. Error in combining the subanswers. By hypothesis the subproblems are the correct ones and their results are correct. Therefore the subresults are incorrectly synthesized into a total answer.

Of course, finding the breakdown from the history may not always be so simple, since recursive programs are not always structured so cleanly. If a program handles a subproblem "inline", it will be a little harder to scan the history for the subproblem's REQUEST and REPLY messages, since they will not be as distinctly marked as function calls are.

Let us put a synthesis bug into our substitute function. If we "forget" the

UNPACK operator from the second sequence (the place is underlined below):

```
(define (substitute =x =y =z)
 (z =>
  (CASES (=> y x)
         (=> [=first !=rest]
             [(substitute x y first) _(substitute x y rest)])
         (else z)) ))
```

the program will incorrectly synthesize the two subproblems `(substitute x y first)` and `(substitute x y rest)` into a complete answer. Now, if we type:

```
(substitute 20 10 [5 10])
```

we will get the result:

```
[5 [20 []]]
```

rather than the desired:

```
[5 20]
```

When this unexpected answer pops out, we simply switch into debug-mode, and have a look at that last computation. If we search backwards (from the end of the history) for a call to `substitute`, we find the innermost call, namely:

```
REQUEST: substitute <= [20 10 []]
CONTINUATION: c1
```

and we jump forwards to the matching REPLY to see:

```
REPLY: c1 <= []
```

So this innermost call is ok. If we back up to the previous call to `substitute`, we see:

```
REQUEST: substitute <= [20 10 10]
CONTINUATION: c2
```

and the matching:

REPLY: c_2 <= 20

which is good. These two calls are at the lowest level of recursion, since neither has a non-empty sequence as the third argument. Back up one call, and up a level of recursion, to find:

```
REQUEST: substitute <= [20 10 [10]]
CONTINUATION: c3
```

with its corresponding REPLY:

```
REPLY: c3 <= [20 []]
```

Aha! We now know that substitute fails to combine its (correct) subanswers properly. We can either look at the code or single-step through the events of this call to find the bug -- the missing "!". If there is some confusion about the temporal relations between the REQUESTs and REPLYs shown above, just make a trace of that part of the computation:

```
REQUEST: substitute <= [20 10 [10]]
CONTINUATION: c3

  REQUEST: substitute <= [20 10 10]
  CONTINUATION: c2

  REPLY: c2 <= 20

  REQUEST: substitute <= [20 10 []]
  CONTINUATION: c1

  REPLY: c1 <= []

REPLY: c3 <= [20 []]
```

DEBUGGING CO-ROUTINES

One can easily dream up analagous bugs exemplifying recursion termination errors and problem subdivision errors. More interesting here is the use of the history with co-routine structured programs. Scanning through the history with the appropriate commands separates out messages for the different co-routines.

Consider the following program, implemented with DELAYs, that computes prime numbers by a sieve algorithm:

```
(define (integers-above =i)
 [(+ i 1) !(delay (integers-above (+ i 1)))])

(define (delete-multiples =n [=first !=rest])
 ((remainder first n) =>
  (CASES (=> 0
             (delete-multiples n rest))
         (else
             [first !(delay (delete-multiples n rest))])) ))

(define (sieve [=first !=rest])
 [first !(delay (sieve (delete-multiples first rest)))])

(primes <- (sieve (integers-above 1)))
```

The sequence primes is an infinite sequence, but it doesn't take long to do the assignment on the last line! This is due to the DELAY feature. DELAY delays the computation within it until the result is actually needed. This means the infinite sequences (integers-above 1) and primes are computed incrementally (and memoized). The functions integers-above, delete-multiples, and sieve are co-routines.

The function integers-above generates an infinite sequence of integers, starting with the successor of its argument. For example:

```
(integers-above -1)
```

when evaluated, will print as:

```
[0 1 2 3 4 5 ...]
```

The source of the ellipsis ("...") is the PLASMA pretty-printer. The pretty-printer abbreviates structured objects in this way when they get longer than a preset length. This is to allow large expressions to fit on the screen, especially infinite ones (see [Downey, 1976]).

In any case, it is important to realize that the sequence above did not get computed out further than its first element until the printer started to print it. Because of this interaction between the printer and delays, it is important that the printing of events from the history or from stepping through a computation not cause any delays to be expanded ("undelayed"). Since the first unexpanded delay in such a sequence has been evaluated (i.e., an environment has been instantiated), the printer prints it as it appears in the source code, except free identifiers within are replaced with their values in that environment.

One way to watch the expansion of the delayed sequence above is to set a breakpoint on `integers-above`, and evaluate `(integers-above -1)`. After a REQUEST to `integers-above` comes, skip forward (computing) to the matching REPLY. A partial trace made in this fashion would look like this:

```
REQUEST: integers-above <= [-1]
CONTINUATION: c1

REPLY: c1 <= [0 !(delay (integers-above (+ -1 1)))]

REQUEST: integers-above <= [0]
CONTINUATION: c2

REPLY: c2 <= [1 !(delay (integers-above (+ 0 1)))]

REQUEST: integers-above <= [1]
CONTINUATION: c3

REPLY: c3 <= [2 !(delay (integers-above (+ 1 0)))]
```

Now we can see why integers-above is a co-routine: it returns an answer before completing its full task.

The functions delete-multiples and sieve are slightly more complex. The (remainder first n) is just the remainder of dividing first by n. The result of (delete-multiples int seq) for integer int and sequence seq is a new sequence like seq, with all the multiples of int deleted. Since this is incrementally computed, the only remainders computed are those that are actually needed. Finally, the co-routine sieve keeps the first element of a sequence (declares it prime), removes all the multiples of the first element from the rest, and sieves that result.

Now, set breakpoints on all three of the above functions, evaluate primes, and follow part of the computation in the same way as above:

```
REQUEST: integers-above <= [1]
CONTINUATION: c4

REPLY: c4 <= [2 !(delay (integers-above (+ 1 1)))]

REQUEST: sieve <= [2 !(delay (integers-above (+ 1 1)))]
CONTINUATION: c5

REPLY: c5 <= [2 !(delay (sieve (delete-multiples 2
                  [!(delay (integers-above (+ 1 1)))]]]]]]]]

REQUEST: delete-multiples <= [2 [!(delay (integers-above (+ 1 1)))]]
CONTINUATION: c6

  REQUEST: integers-above <= [2]
  CONTINUATION: c7

  REPLY: c7 <= [3 !(delay (integers-above (+ 2 1)))]

REPLY: c6 <= [3 !(delay (delete-multiples 2
                  [!(delay (integers-above (+ 2 1)))]]]]

REQUEST: sieve <= [3 !(delay (delete-multiples 2
                     [!(delay (integers-above (+ 2 1)))]]]]
CONTINUATION: c8

REPLY: c8 <= [3 !(delay (sieve (delete-multiples 3
                  [!(delay (delete-multiples 2
                     [!(delay (integers-above (+ 2 1)))]]]]]]]]]]]
```

All that just to get the first two prime numbers! UNPACKed DELAYS are not exactly pellucid, so let's take advantage of hindsight and look at the same computation from the point of view of the history, after the delays have been expanded out several times:

REQUEST: integers-above <= [1]
CONTINUATION: c_4

REPLY: c_4 <= [2 3 4 5 6 7 ...]

REQUEST: sieve <= [2 3 4 5 6 7 ...]
CONTINUATION: c_5

REPLY: c_5 <= [2 3 5 7 11 13 ...]

REQUEST: delete-multiples <= [2 [3 4 5 6 7 8 ...]]
CONTINUATION: c_6

REQUEST: integers-above <= [2]
CONTINUATION: c_7

REPLY: c_7 <= [3 4 5 6 7 8 ...]

REPLY: c_6 <= [3 5 7 9 11 13 ...]

REQUEST: sieve <= [3 5 7 9 11 13 ...]
CONTINUATION: c_8

REPLY: c_8 <= [3 5 7 11 13 17 ...]

While this trace may seem strange in its order of computation, it is easier to see that each co-routine is executing correctly. A trace of a single routine in this hindsight fashion, say, of delete-multiples, should be most illuminating:

```
REQUEST: delete-multiples <= [2 [3 4 5 6 7 8...]]
CONTINUATION: c6

REPLY: c6 <= [3 5 7 9 11 13 ...]

REQUEST: delete-multiples <= [3 [5 7 9 11 13 15 ...]]
CONTINUATION: c9

  REQUEST: delete-multiples <= [2 [4 5 6 7 8 9 ...]]
  CONTINUATION: c10

    REQUEST: delete-multiples <= [2 [5 6 7 8 9 10 ...]]
    CONTINUATION: c10

  REPLY: c10 <= [5 7 9 11 13 15 ...]

REPLY: c9 <= [5 7 11 13 17 19 ...]
```

Now that we can follow the execution of a co-routine in an understandable manner, most of the previously mentioned methods for finding bugs can be applied.

CONCLUSIONS

Historically, debugging tools in programming languages have varied in power from nearly useless (e.g., core dumps) to reasonably useful (e.g., stack examining features and single-steppers); the practical ones being confined largely to interpreted languages. Since PLASMA is a message-passing language, a recorded history of the events of a computation, together with a large number of useful ways to examine this history, provides the user with an exceptionally flexible and powerful means of debugging his PLASMA programs.

SUGGESTIONS FOR FUTURE WORK

There is a need in PLASMA and other languages for a general method of structuring error messages. It is necessary to distinguish between different kinds of execution errors, and, depending upon the context, handle them in different ways. One kind of error "should never happen"; for example, a syntax error. Usually this kind of error must be handled manually. However, there are times (e.g., when reading in a file of function definitions) that this kind of error should be handled automatically.

Another kind of error is sometimes anticipated, it signals the failure of some algorithm looking for an answer. A failure message should bypass intermediate levels of program structure and reach the routine that knows what to do next in the situation.

The problem is not the number or meaning of error messages, but determining how to handle them. Since the handling of different kinds of errors should be dynamic, and is just as dependent upon the context of the error as on the source of the error, the structure of error messages and the algorithm for deciding how to handle them are very important issues.

Another area with room for future work is debugging programs with side-effects. Although the history-keeper remembers side-effects, and the history-motion commands restore side-effects, it is not clear how, in general, to use these features to good advantage. There is a slightly fuzzy but important conceptual difference during debugging between routines that directly use side-effects (e.g., CONS as defined earlier), and routines that just make use of such subroutines. In the former case, all of the concerns local to the side-effects must be debugged. In the latter case, the issue is programs that run in an environment with

side-effects.

REFERENCES

Downey, Thomas S. "A Pretty-Printer for PLASMA".

Unpublished bachelor's thesis, Dept. of Electrical Engineering and Computer Science,

M.I.T., Cambridge, Mass. May, 1976

Grief, I. and Hewitt, C. "Actor Semantics of PLANNER-73".

Proceedings of ACM SIGPLAN-SIGACT Conference,

Palo Alto, California. January, 1975

Hewitt, Carl "Viewing Control Structures as Patterns of Passing Messages".

A.I. Memo 410, M.I.T. Artificial Intelligence Laboratory,

M.I.T., Cambridge, Mass. December 1976

Hewitt, Carl "A PLASMA Primer".

Draft paper, M.I.T. Artificial Intelligence Laboratory,

M.I.T., Cambridge, Mass. 1974