An Operational View of Lucid

by

Brian Finch

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Research Report CS-81-37
December 1981

ABSTRACT

We examine a number of Lucid programs, and consider their semantics from an operational view which permits multiple simultaneous or concurrent processes to mutually communicate, as well as create new processes.

INTRODUCTION

Computer science has always suffered from a certain amount of factionalism which has, more often than not, served to impede progress rather than to encourage it. For example, perhaps the most damaging gap occurs between the fields of hardware design and software design. In particular, the discipline of programming language design is one which stands to benefit greatly from a more intimate relationship here than it has enjoyed in the past. This is not to be taken as advocating a return to assembler-language programming, quite the contrary, rather it should be understood as a plea for increased dialogue, a symbiosis of sorts, between these specialists.

Of course, there has always been a certain amount of interaction at this level; for instance the introduction of tasks and events in PL/1 [14], or of coroutines in SIMULA67 [14] clearly indicate that language designers were indeed aware of hardware developments. Going back even further, it is easy to see where the first GOTO statement originated. Similarly, in the other direction, we see that most computers now have indivisible hardware instructions allowing the implementation of semaphores for synchronizing concurrent processes running on a single processor. This is an example of software systems considerations influencing hardware design.

The above examples notwithstanding, the gaps between languages and hardware which remain are significant. Although multi-processor architectures have been extant since at least 1962 [13], it is remarkable that it has been only recently that serious efforts have been made to develop languages for controlling these machines in a rational and well-structured manner. Similarly, the message-passing approach, which is particularly appropriate in computer network environments, is just now coming into its own [15]. Less optimistically, computers based on associative networks can likely be expected to remain in obscurity for some time yet [13].

At first glance, these machine considerations might seem unnecessary from the point of view of a language designer: after all, we know that any "reasonable" computer that we might deal with can be considered as equivalent to a Turing machine, so if our language is only capable of expressing computations executable by a Turing machine, then there will be a total recursive function ("compiler") mapping our programming language onto our "reasonable" computer.

In the case of Lucid, the subject of this paper, the above observation is not immediately applicable: the basic data objects in Lucid are infinite, so our computations will likely never terminate, and thinking in terms of Turing machines is not strictly correct. In many implementations of Lucid, however, values of infinite objects are calculated one component at a time on a demand basis only, so the Turing relationship is approximately correct.

Unfortunately, the types of computations which result at the machine level under this kind of interpretive scheme are often utterly unlike those which the programmer envisioned when writing the program. In and of itself, this is perfectly acceptable, except that in making the necessary transformations, a compiler might well have to take a conceptually simple and elegant description of a computational process, and hammer it into an awkward and unnatural one. It might be argued that this is what compilers are for, but a more convincing argument would point out that, on the contrary, the reason for having available unusual computer architectures, is to save compilers the bother.

In this paper, therefore, we will try to view several Lucid programs in operational terms which are significantly different from what existing compilers and interpreters produce, which use an uncommon architecture, and which are arguably more natural expressions of the computational process.

MOTIVATION

It has been pointed out [7] that in order to correctly execute Lucid programs, some sort of (real or simulated) parallel activity is needed. In a single-processor environment, this is, of course, quite difficult to implement, since it involves "dove-tailing" multiple computations to occur concurrently. In a multi-processor environment, assuming the presence of an amenable operating system, it should be no problem at all. (Most multi-programmed, single CPU operating systems are also quite capable of this sort of task, but the overhead involved can slow things down significantly. Besides, using multiple processors for performing parallel activities is so sensible.)

In a theoretic description of the operational semantics of Lucid, Farah [12] describes a tree-manipulation system in which the parse tree of a program is transformed non-deterministically into an "answer". Here there is an allowance for slightly more parallelism than is required by the semantics.

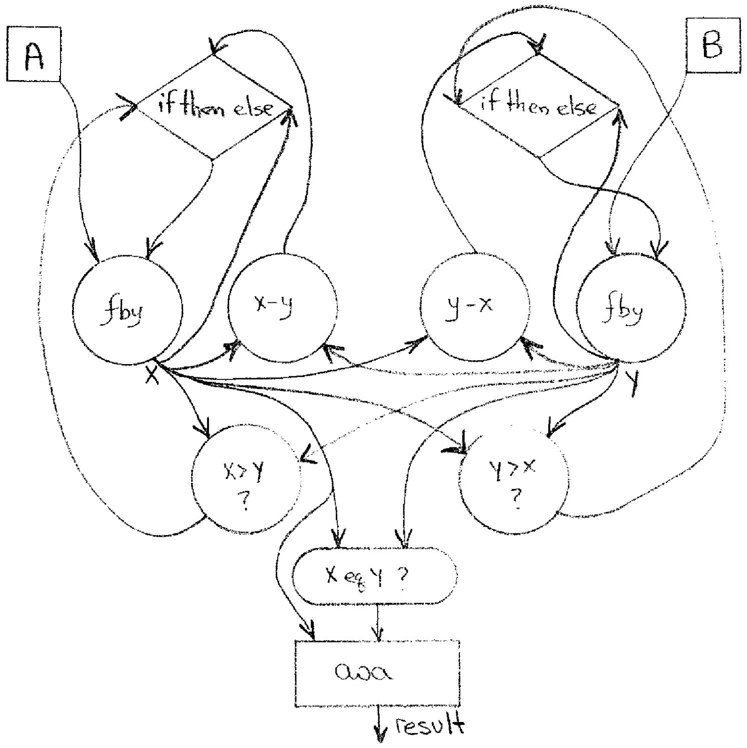
In what follows, we will take for granted the "necessary" parallelism (essentially, in evaluating the logical "AND" and "OR" operations), and add to that any finite amount of parallelism that is deemed reasonable or appropriate to the program at hand, when examining the multi-processor approach.

GCD

No paper on programming language techniques would be complete without an example of a program for calculating the greatest common divisor of two integers. Consider the following Lucid program, where A and B can be thought of as constant sequences global to the <u>valof</u> phrase:

```
valof
          X = A fby if X > Y then X - Y else X
          Y = B fby if Y > X then Y - X else Y
     result = X asa X eq Y
end.
```

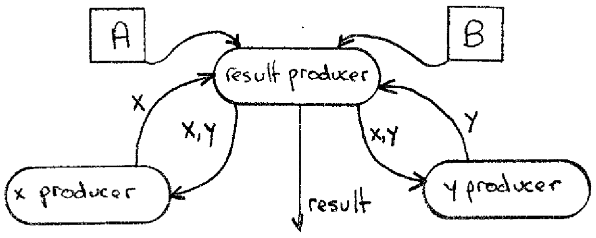
An appealing model for this computation can be got with the help of data-flow networks. The obvious translation of the above program would give a net something like this:



There are two problems with this representation. The first is that, in general, the if-then-else nodes cannot be expected to perform as required. They must consume all three of their arguments at the same rate, so if the input which is not selected is a non-terminating computation, for instance, subsequent computations will likely be wrong.

The second problem is an aesthetic one. The original program described the computation as a set of three equations which defined three objects (X, Y, result), while the data-flow net does the same thing, by way of a dozen nodes and lines all over the place. Clearly, a program of reasonable size could generate an impossibly large network, which would be incomprehensible.

The reason for this explosion in size of the data-flow model is that every single operation in the original program must now be represented by its own node, with its own input and output paths as well. What would be useful at this stage would be a mechanism for hiding all the low-level detail, without obscuring the intent. In this case, a diagram like this would be appropriate:



This expresses the nature of the computation very broadly, without including too much detail. A more succinct and complete representation of the details of the computation could be given as follows*:

```
module producer
    while true do
          (I,J) := receive(parent);
          if I > J then send(parent, I - J);
```

*For a description of the language used here, see the Appendix.

```
                    else send(parent, I);
               fi
            end while
        end module

        main(A, B)
            X_Prod := instantiate producer;
            send(X_Prod, (A, B));
            X := receive(X_Prod);
            Y_Prod := instantiate producer;
            Y := receive(Y_Prod);
            send(Y_Prod, (B, A));
            while true do
                   if X = Y then print("result=", X)
                           else send(X_Prod, (X,Y));
                                X := receive(X_Prod);
                                send(Y_Prod, (X,Y));
                                Y := receive(Y_Prod);
                   fi;
            end while
        end.
```

Since the definitions of X and Y in the Lucid program are symmetric, they can be coalesced and represented as a single module in an environment which supports dynamic process creation and message passing.

What is interesting about this latter approach is that, in a sense, it is a data-flow representation of the computation, but on more of a macroscopic level than pure data-flow.

Recursive Factorial

This is a program which was presented in [1] as being anomalous:

```
valof
        N = 7 fby N - 1
    result = if N <= 1 then 1
                          else N * next result
end.
```

It was pointed out in [1] that this program can be readily translated into neither a set of simple loops nor a data-flow network. The reason for this is that the program calculated its answer, <7!, 6!, ..., 2!, 1!, 1, 1, 1, ...> from the inside out - the ones can be found first, and then the first seven elements are got by recursing backwards to the head of the list.

Consider the following translation of the above program:

```
module N_Prod(N)
     while true do
         send(parent, N)
         N := N - 1;
     end while
end
```

```
module result_prod(Y)
     instantiate N_Prod(Y);
     M := receive(N_Prod);
     if M <= 1 then send(parent, 1)
               else Y := receive(N_Prod);
                    instantiate result_prod(Y);
                    send(parent, M*receive(result_prod));
     fi;
end

main()
     instantiate N_Prod(7);
     while true do
         X := receive(N_Prod);
         instantiate result_prod(X);
         print("result=", receive(result_prod));
     end while
end.
```

The interesting aspect of this interpretation is the way in which the module "result_prod", when at time t, will make a copy of itself and its environment (N_Prod), if it needs the value of itself at time t+1. This is, in fact, a thinly disguised version of a demand-driven calculation, but one which leaves the hierarchy of requests implicit.

As such, this solution suffers from the same problem as would any "unintelligent" demand-driven scheme: it ends up re-calculating several values more often than necessary. In order to evaluate "result" at time 0, it must first find "result" at time 1 and so on, but as it stands here, this program, having successfully determined the first "result", would than start from scratch to find "result" at time 1, and again at time 2, and so on.

It is conceivable that in the appropriate enviroment, a compiler could be capableof making the necessary optimizations to avoid this kind of redundant computation.

A Simple Luswim Program: Sum of Powers.

Here is an example from [3] which is a pure Luswim program, as all of the variables are elementary. This program calculates the sequence which is everywhere "the sum from i=1 to 6 (i**i)"

```
valof
        I = 1 fby I + 1
        S = 1 fby S + next M
        M = valof
                   K = 1 fby K + 1
                   P = 1 fby I * P
              result = P asa K eq I
            end
   result = S asa I eq 6
end.
```

This program is operationally very simple, since its main mechanism is so familiar. The mechanism is that of the subroutine call, which corresponds to what one expects of the valof phrase defining the value of M. Since the global of the inner phrase (I) is elementary, the locals of the phrase (K, P, result) are constrained to "restart" whenever "time" in the outer environment advances. Thus, we would expect the computation to proceed like this:

```
function prod_m(A);
   K := 1;
   P := A;
   while K ne A do
       P := P * A;
       K := K + 1;
   end while
   return(P);
end.

main()
   I := 1;
   S := 1;
   M := prod_m(I);
   while true do
       if I eq 6 then print("result=", S);
                 else M := prod_m(I);
                      S := S + M;
                      I := I + 1;
       fi;
   end while
end
```

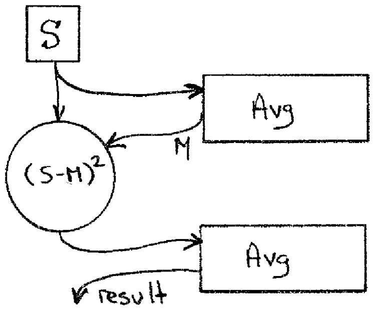
This demonstrates the simple relationship between Luswim and ordinary imperative programming languages, recursion notwithstanding, which should make the compiling of Luswim programs a very easy and straightforward task.

A ULU PROGRAM: VARIANCES

In ULU, things are not quite as easy for us as in Luswim. Consider this program from [3], which calculates the variance of the first 10 elements of the free variable S:

```
valof
        Avg(X) = valof
                        S = X fby S + next X
                        N = 1 fby N + 1
                   result = S / N
                 end
             M = Avg(S) asa I eq 10
             I = 1 fby I + 1
        result = Avg((S - M)**2) asa I eq 10
end.
```

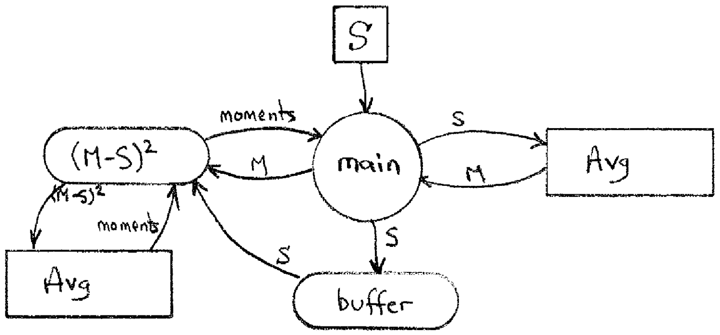
The easiest way to see the basic problem here is to look at the skeleton of a data-flow network for this program. Without going into unnecessary detail, we would find a network that looks something like this:



The difficulty is that the "S" node is required to pump its first ten values into the "Avg" box on the right before the first "M" is produced, and thus before the first (S - M)**2 can be calculated. This clearly is a buffering problem, something that data-flow networks cannot readily handle.

A discussion of this program in terms of coroutines appears in [3], although the aspect of buffering is not considered. Often one can finesse this sort of problem by assuming one's compiler to be smart enough, but in this case it is instructive to do otherwise.

The basic structure of the computation defined by this program can be viewed thus:



We assume that we will have one sequential source for our "S" values, i.e. that, having accessed S at time t, the values of S at times 0 through t-1 are gone forever. A much more important assumption, however, is that in implementing the communications between processes, "unblocked sends" be permitted. In practice, this is not an unusual feature [15].

These unblocked sends show up in the movement of S values from "main" through "buffer" to "diffsq" (Actually, the situation is totally symmetric, so it would be perfectly correct to send the S's directly to "diffsq" and divert the M's via "buffer" instead). This mechanism allows the production of S values to become arbitrarily "out of step" with their consumption.

(In fact, the entire module "buffer" is really not necessary here; the message buffering takes place (conceptually) in transmission, rather than in any routine's data space. "buffer" is included here mainly for clarity - without it "diffsq" would have to read two types of data (M's and S's) which are unsynchronized, and which originate in the same module (main). In such a case, care would have to be taken to ensure that the value received is indeed from the intended variable. This would depend to a great extent on the design of our message-passing language, which is not our concern here.)

Here is a program to implement the above process structure.

```
module average
     S := 0;
     N := 1;
     while true do
          S := receive(parent) + S;
          send(parent, S/N);
          N := N + 1;
     end while
end

module buffer
     while true do
          send(diffsq, receive(parent));
     end while
end

module diffsq
     MOM := instantiate average;
     while true do
          M := receive(parent);
          S := receive(buffer);
          send(MOM, (S - M)**2);
          send(parent, receive(MOM));
     end while
end

main()
     RA := instantiate average;
```

```
      instantiate buffer;

      instantiate diffsq;

      I := 1;

      while true do

          S := receive(S_Prod);

          if I < 10 then send(buffer, S);

                         send(RA, S);

                         AV := receive(RA);

                     else if I = 10 then

                         AV := receive(RA);

                         send(diffsq,AV);

                         answer := receive(diffsq);

                     else

                         print("result =", answer);

                     fi

           fi

          i:= i + 1;

      end while

end.
```

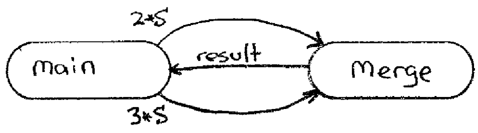
Lucid - Merge Function

In the previous example, the degree to which modules could become "out of step" with each other was determined parametrically ("asa I eq 10"), and was a finite and bounded amount. What should we do when this quantity is essentially unbounded? No computer system could allow an indefinite number of unread messages sent unblocked - all the space in the file system would fill up, and the computation would grind to a halt.

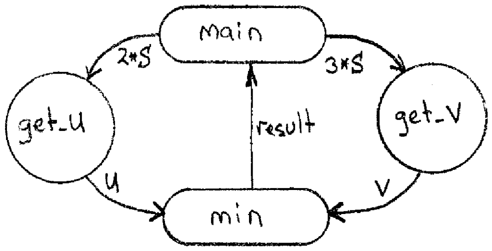
The Lucid operation which is most prone to this kind of behaviour is the upon function, and by now a classic example of upon in action is the following program, which determines the sequence of all numbers of the form (2**i)*(3**j), i>=0, j>=0, in ascending order:

```
valof
    merge(A, B) = valof
                          U = A upon (U eq M)
                          V = B upon (V eq M)
                          M = min(U, V)
                     result = M
                  end
              S = 1 fby merge(2*S, 3*S)
         result = S
 end
```

If we look at a diagram of the movement of data in the outer environment, it is not very instructive:



But, if we look at the "merge" function in more detail, we can see more clearly what is going on:



When we have loops in these diagrams, which can be thought of as showing result dependencies in calculations, we might expect to have to worry about the possibility of deadlock occurring. However, this situation arises only when the semantics of Lucid tells us that something is not properly defined, as in:

```
I = next I fby first I,
```

which defines a constant sequence which is everywhere 'anything'. We wouldn't expect programs with such statements to be meaningful, so it is acceptable that our message-passing approach doesn't work here.

Here is a full program to implement the "merge" program above:

```
module get
     X := receive(main);
     while true do
          send(min, X);
          M := receive(min);
          if M = X then X := receive(main) fi;
     end while
end

module min
     while true do
           A := receive(get_U);
           B := receive(get_V);
           Y := if A<B then A else B fi;
           send(get_U, get_V, main, Y);
     end while
end

main()
     get_U := instantiate get;
     get_V := instantiate get;
     Z := 1;
     while true do
```

```
                print("result=", Z);
                send(get_U, 2*Z);
                send(get_V, 3*Z);
                Z := receive(min);
            end while
        end.
```

Lucid - Running Averages of Square Root

In this example we will examine a program which uses both elementary and non-elementary functions. Up until this point, we have been careful to look at programs which exercise only one of Lucid's "unusual" features at a time, so it is worthwhile to see what happens when we combine several in a single program. It is not unreasonable to expect that such a combination would lead to an inpenetrable mess, but as we shall see, this is not the case.

Here is the Lucid program in question:

```
valof
    Avg(A) = valof
                        S = A fby S + next A
                        I = 1 fby I + 1
                   result = S / I
               end
   Sqrt(N) = valof
                         X = 0 fby X + 1
                         Y = 0 fby Y + (2*X) + 1
                    result = X - 1 asa Y > N
               end
     result = Avg(Sqrt(Y))
end
```

Y is assumed to be a global variable, whose values are first converted into their integer square roots (poin-twise), with those latter then having their running averages calculated.

A message-oriented program could be as follows:

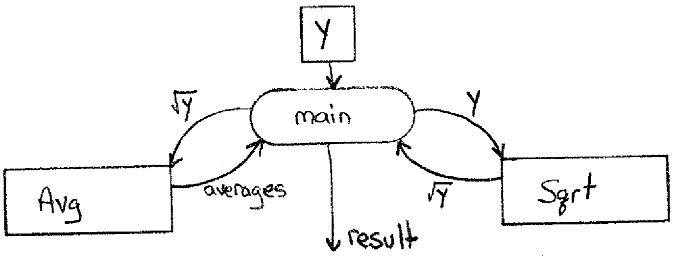
```
module Avg
     I := 1;
     S := 0;
     while true do
          S := receive(main) + S;
          send(main, S/I)
          I := I + 1;
     end while
end

module sqrt
     while true do
          X := 0; Y := 0;
          N := receive(min);
          while Y <= N do
               Y := Y + 2*X + 1;
               X := X + 1;
          end while
          send(main, X - 1);
     end while
 end
```

```
main()
     instantiate avg;
     instantiate sqrt;
     while true do
          Y := receive(Y_Prod);
          send(sqrt, Y);
          send(avg, receive(sqrt));
          print("result=", receive(avg));
     end while
end
```

Notice the difference between "avg" and "sqrt" - since "sqrt" must restart with each new value passed to it, this causes the initializations in the module to be placed within the infinite loop, whereas "avg", which calculates running averages, and thus must "remember" its previous computations, has its initializations outside the loop, so that they are executed once only. This is a very simple description of the difference between elementary and non-elementary functions, one to which any programmer can easily relate.

Again the structure of this computation is very simple:



The fact that "avg" and "sqrt" "work differently" in some sense, is irrelevant at this level of abstraction. We have succeeded in effectively hiding the lower-level details, without obscuring the real nature of the process.

Lucid - Elementary and Non-elementary Parameters

For our final problem, we will consider a function which has both elementary and non-elementary parameters. As we will see, this leads to a message-passing program of rather interesting structure. The program in question is Mom2(X,M), which calculates, at time t, the value of the second moment of the first t+1 values of X about M at time t:

```
Mom2( X , M ) = valof
                   S = T fby S + next T
                   T = (X - M)**2
                   I = 1 fby I + 1
              result = S / I
end.
```

The facts that the first parameter to Mom2 is non-elementary and that the second parameter is, basically tell us that this function must, at time t, be able to access the first t+1 values of the first parameter, but need only be able to see the current value of the second. This is really quite straightforward, except for one point which we have not yet encountered in any of the other programs in this paper: we must be able to handle correctly the situation where some of the values of the second argument to Mom2 are undefined. We must be careful since M at time t being undefined does not imply that result at time greater than t will be undefined, even though result at time t is.

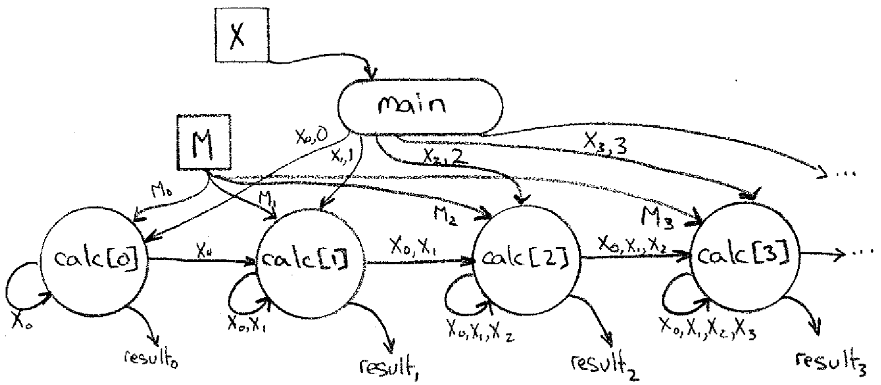
(Note that we have nothing to worry about if some value of X is undefined, since this would cause all subsequent values of the result to be undefined; it is no problem simply because it is so easy to make a program produce no result, as would be required.)

The message-passing solution that we propose for this program is the following:

```
main()
    i := 0;
    while true do
        a := receive(X_prod);
        proc := instantiate calc[i];
        send(proc,i);
        send(proc,a);
        i := i+1;
    end while;
end

calc()
    count := receive(parent);
    for i:= 1 to count do
        a := receive(calc[count-1]);
        send(calc[count],a);
        send(calc[count+1],a);
        end for;
        a := receive(parent);
        send(calc[count], a);
        send(calc[count+1], a);
        b := receive(M_prod);
        sum := 0;
        for i := 0 to count do
            sum := sum + (b - receive(calc[count])) ** 2;
        end for;
        print("result",count,"=", sum/(count+1) );
    end calc.
```

To make this a bit more clear, we can represent the structure of the calculation thus:



If some calc, say calc[i], fails to receive its requested value of M, it will become blocked while waiting for the value that is never sent. This doesn't disturb any other part of the calculation, since calc[i] has already sent along to calc[i+1] all of the values that it was required to. If main fails to receive some X, it will be unable to obtain any subsequent value of X either, so that calc[j], j>=i will not receive their necessary inputs, and hence will not calculate any results.

The solution presented here is clearly not an ideal one. For instance, the trick that calc performs of reading its input and then sending it to itsel is a rather awkward way of reading its input twice. The most important point being made here is that all of the Lucid programs considered here have a reasonably concise and natural expression in what seems to be a substantially different idiom, that of message-oriented programming.

CONCLUSIONS

Although Lucid programs do not translate nicely into many of our standard models of computation, such as iteration, data-flow, recursion, or coroutines, it does seem that they have a simple and natural expression in terms of dynamic process creation and message passing. In fact, the translations of programs presented here are so directly related to the syntax of the original Lucid code that this gives reason to hope that the task of generating such translations could be reasonably easily automated. This is certainly a possible direction for future work.

The greatest problem discovered in the course of this work is representational. When dealing with "standard" computational processes, we have a fairly flexible and well established notation for helping us visualize such computations in the flowchart. We can represent graphically the necessary control structures (iteration, alternation), in a way that seems straightforward and natural. However, when we want to talk about less usual concepts like dynamic process creation, coroutines, and so on, we really don't have the proper tools necessary for concisely describing these kinds of activity. This is perhaps another direction in which more work is needed.

APPENDIX

The language used in this paper for describing message-passing, process-creating programs is basically an extension of Pidgin Algol, with a few new primitives tacked on. The additions consist of the concept of modules, and three commands: instantiate, send and receive. The language owes much to [15].

A module is taken to be a block of code which remains dormant until it is referenced by an instantiate command. The syntax for this command is:

```
[ id := ] instantiate module_name .
```

This command arranges, presumably through the host operating system, for a processor to be allocated and run on the code embodied in the module. This process is known by the name of the module or optionally by the assigned identifier (in such cases when the module has several instances, to avoid ambiguity), and knows its creator by the name "parent".

The send and receive commands are self-evident from their syntaxes:

```
send( to_process_name {, to_process_name}* , value )

id := receive( from_process_name )
```

or

```
receive( from_process_name )
```

The latter version of receive allows the received value to be used directly in an expression.

REFERENCES

[1] Ashcroft, E.A., and Wadge, W.W., Lucid, A Nonprocedural Language with Iteration, CACM, June 1977, pp. 519-526.

[2] Ashcroft, Ed, and Wadge, Bill, A Logical Programming Language, Research Report CS-79-20, University of Waterloo, June 1979.

[3] Ashcroft, Ed, and Wadge, Bill, Structured Lucid, Research Report CS-79-21, University of Waterloo, June 1979.

[4] Ashcroft, E.A., and Wadge, W.W., Some Common Misconceptions about Lucid, Research Report CS-79-38, University of Waterloo, Dec. 1979.

[5] Ashcroft, E.A., and Wadge, W.W., Lucid, a Formal System for Writing and Proving Programs, SIAM J. Comput., Sept. 1976 pp. 336-354.

[6] Wadge, W., Away from the Operational View of Computer Science, unpublished notes, University of Warwick.

[7] Cargill, T.A., Deterministic Operational Semantics for Lucid, unpublished report, University of Waterloo, 1977.

[8] Ostrum, C.B., Luthid0.0 Preliminary Reference Manual and Report, unpublished paper, University of Waterloo, 1980.

[9] Friedman, Daniel P., and Wise, David S., A Note on Conditional Expressions, CACM, Nov. 1978, pp. 931-933.

[10] Vuillemin, Jean, Correct and Optimal Implementations of Recursion in a Simple Programming Language, JCSS, 9, 1974, pp. 332-354.

[11] Culik, Karel, and Farah, Mansour, Linked Forest Manipulation Systems -- A Tool for Computational Semantics, Research Report CS-77-18, University of Waterloo, 1977.

[12] Farah, Mansour, Correctness of a Lucid Interpreter Based on Linked Forest Manipulation Systems, Intern. J. Computer Math., Section A, 8, 1980, pp. 3-26.

[13] Handler, W., ed., Computer Architecture, Springer-Verlag, Berlin, 1976.

[14] Pratt, T.W., Programming Language Design and Implementation, Prentice-Hall, Englewood Cliffs, N.J., 1975.

[15] Koch, Andres, MENYMA, Design and Implemantation of a Message Oriented Language, Masters Essay, University of Waterloo, Dec. 1980.