SOME COMMON MISCONCEPTIONS ABOUT LUCID

E.A. Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

W.W. Wadge
Department of Computer Science
University of Warwick
Coventry, England

Research Report CS-79-38
December 1979

SOME COMMON MISCONCEPTIONS ABOUT LUCID

Ed Ashcroft
Computer Science Department
University of Waterloo
Waterloo, Canada

Bill Wadge
Computer Science Department
University of Warwick
Coventry, England

Abstract

This paper attempts to clear up several misconceptions about the language Lucid. In the process we claim that Lucid is in fact a real programming lanugage, and indicate various ways in which implementations might be feasible.

0. *Introduction*

Lucid [3] began to be developed five years ago, by the authors, as a language in which it would be easy and straightforward to prove assertions about programs. Rather than devote our efforts to developing more powerful tools to verify programs in existing languages, or attempt to modify existing languages to make them more verifiable, we concentrated our efforts on re-assessing what constitutes a programming language and on inventing a completely new language based on mathematical principles. This language, Lucid, has achieved a certain amount of success, for example attracting some interest from people doing research in parallel computation, but, in spite of this, Lucid is not really taken seriously as a practical programming language. This is partly because there is, as yet, no really practical implementation, and partly because the language is still being developed. Moreover, the current lack of a practical implementation is caused by the fact that standard compiling or interpreting techniques cannot be used or are not relevant, and this had led some to conclude that Lucid is so unusual that it is inherently impractical (although paradoxically, there are others who say that it adds nothing new!). We feel that these views, while understandable, are wrong, and are based on simple misconceptions, for which we ourselves are partly to blame.

In this paper we therefore will try to clear up some of these misconceptions, answering six of them in detail. These are not just misconceptions about Lucid, but are misconceptions about all nonprocedural languages and about more general issues, particularly the relationship (actual and desired) between operational and mathematical semantics.

We will list what we see as the main misconceptions, following each one by a discussion which, hopefully, clarifies the issue. It is assumed that the reader has some familiarity with [4] and [5].

1. *Lucid is just another purely recursive language*

Purely recursive languages have been around for a long time, for example Pure LISP [13] and ISWIM [11], and, of course, Kleene's recursive equations [10]. All have the same power in that they can compute (assuming the availability of arithmetic operators) any partial recursive function. They do this without using assignment statements or indeed any forms of command that cause changes in the values of variables, and the languages are therefore referentially transparent. (This means, to quote Stoy [16], "The only thing which matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same whenever it occurs.") This gives such languages very desirable mathematical properties. Lucid is such a language, one in which expressions denote infinite sequences of data objects.

Apart from the choice of the data domain, the difference between Lucid and the others is purely a matter of form; but for programming languages, which are, of course, tools, form is very important. The data objects and operations in Lucid are such that many Lucid programs can be understood as programs using a general form of iteration. We feel that most programmers find iterative algorithms, when appropriate, to be more understandable than recursive ones, and we feel that implementations which use iteration (when appropriate) can be extremely effective.

For example, compare the following ISWIM program for fast exponentiation, to compute x^n,

```
pow(n,x) = aux(n,1,x)
    wherec
        aux(k,y,p) = if k eq 0 then y
                     else if even(k) then
                        aux(k div 2,y,p·p)
                     else aux(k div 2,y·p,p·p)
    end.
```

with the corresponding Lucid program:

```
valof
    first k = n
    first p = x
    first y = 1
    next  y = if even(k) then y else y·p
    next  k = k div 2
    next  p = p·p
    result = y asa k eq 0
end.
```

(The operation asa is as soon as and the operation div is integer division.) It is obvious from the Lucid program that an iterative algorithm is being used, and the nature of the algorithm is clearer.

(The Lucid program can be written more concisely using the operation fby or followed by:

```
valof
    k = n fby k div 2
    p = x fby p·p
    y = 1 fby if even(k) then y else y·p
    result = y asa k eq 0
end.
```

There seems to be a paradox here; Lucid claims to be a mathematical language but one of its big advantages (iteration) is concerned with operational matters. This is another misconception - operational *thinking* and can be very useful as a programming aid and iteration is very useful, both conceptionally and practically. The nature of the language is still mathematical. The problem with purely recursive languages is that they are too general and use only a few basic, but very powerful, tools. Lucid is richer and allows one to express simple algorithms in a simple way.

Purely recursive languages can imitate Lucid by defining a separate function for each Lucid variable, this function taking one argument, a special "time parameter".* Thus, corresponding to the previous Lucid program, we would have four functions, including

```
y(i) = if i eq 0 then 1 else if even(k(i-1)) then y(i-1)
                                else y(i-1)·p(i-1)
```

and

```
result(i) = y(f(0)) wherec
                f(j) = if k(j) eq 0 then j
                                    else f(j+1)
            end.
```

This is clearly syntactically more complicated than the Lucid program and is using an unnecessarily powerful tool (recursion) to express a simple idea (iteration). If this "time parameter" idea is used only to model iteration then the function definitions will have restricted syntactic forms, suggesting that they could be generated by a preprocessor from some less clumsy notation. Lucid is exactly such a notation, and it has the advantage that it has its own semantics and enjoys all the

* A variant of the "time parameter" idea is found in the work of Arsac [1], who independently developed a language based on recurrence equations.

desirable properties that the recursive language has. Moreover, if the "time parameters" are used in non-standard ways, for example to get at the "past history" of variables, the same effects can usually be obtained in Lucid by using new operations, like hitherto. In other words, new operational features are obtained by extending a simple language rather than weakening the restrictions on a more powerful one.

2. *Lucid is just another single-assignment/data-flow/coroutine language*

The idea of single-assignment languages [17,2] predates Lucid. Basically, single-assignment languages are conventional iterative imperative languages with assignment, in which syntactic restrictions are imposed. These restrict the use of assignment statements so that for each loop and each variable changed by the loop there can be only one assignment to the variable before entry to the loop and only one inside the body of the loop. There are also restrictions on conditional statements so that the same variables are assigned to in the two branches of the conditional.

Single assignment languages vary, but in general each single-assignment program corresponds to a simple Lucid program in a fairly direct way (especially if we write the Lucid program using first and next rather than fby). These corresponding Lucid programs will themselves obey certain restrictions (no Lucid operations on the right-hand-sides of equations, restricted use of asa, no defined functions, etc.) so both these and single assignment languages are special cases of more general languages. In the case of single-assignment languages however, removing these restrictions is a step backwards to general imperative languages, whereas with Lucid, removing these restrictions retains the fundamental properties of the simple Lucid programs (for example, referential transparency) and gives a more powerful language. New operations, such as fby, upon and whenever, (not corresponding to any imperative construct) can be added, as can user-defined functions, even non-elementary and non-pointwise ones. Lucid is far more than a single-assignment language.

Lucid functions can often be understood operationally as defining coroutines, as indicated in [5]. Some imperative languages (for example EPL [12] and Kahn & McQueen's language [9]) have been designed to give the user the 'facility' to establish coroutines, by having "actors" pass "messages" to each other. Some people have jumped to the conclusion that Lucid is just another such coroutine language. The difference is that Lucid is defined mathematically and some form of message passing can be used to implement certain Lucid programs - although other methods (e.g. compiling) could be used as well.

The coroutine languages were operationally motivated and the user has the burden of creating actors and sending and receiving messages. In Lucid these simply correspond to defining functions and using them in the conventional way, giving them expressions as arguments and composing them to form other expressions. For example, we can write two Lucid functions, *produce* and *consume*, such that *consume*$(\underset{\sim}{X})$ gives us an infinite stream of results, where $\underset{\sim}{X}$ is an infinite stream of 'resources' (many resources may have to be consumed to provide one

result, or one resource may provide many results), and *produce*(*Y*) gives us an infinite stream of resources, where *Y* is an infinite stream of external data items (once again, one data item may produce many resources, or many data items may be needed to produce one resource). These functions can be interpreted as coroutines, and the expression *consume*(*produce*(*Y*)) corresponds to linking them together in the usual way. We can even feed the results of consumption back as the data items that determine the resources that are produced, as follows:

X = *consume*(*produce*(*Y* fby *X*)).

Here the first data item in *Y* sets the whole thing off, subsequent inputs to *produce* being the results of *consume*. Clearly this might give us deadlock, which corresponds to the value of *X*, according to the mathematical semantics, being undefined ($\perp$) at some point.

Many meaningful Lucid functions can not be viewed as coroutines, unless we have a 'restart' facility. Consider, for example, the following function *Mom2*

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

which is such that *Mom2*(*A*,*N*) at time t is the second moment, about the value of *N* at time t, of the first t+1 values of *A*.* (If we have a function *Avg* that produces the stream of running averages of its input stream, then *Mom2*(*A*,*Avg*(*A*)) will be the stream of running variances of the stream *A*.) *Mom2*(*A*,*N*) can be interpreted as a coroutine, which is being fed with two streams *A* and *N*, but only if for each value of *N* the coroutine can internally restart the stream *A* in order to subtract this latest value of *N* from the latest and *all the previous* values of *A* (and square the results and take the average). Lucid is clearly more than a normal coroutine language.

Wadge [8] has indicated how some simple Lucid programs (which are just sets of equations) can be implemented as data-flow networks. In fact all such networks correspond to such simple Lucid programs. The fact that there are Lucid programs that are not equivalent to such simple ones means that Lucid is more than a language for programming data-flow machines. This subject is considered more in section 5.

3. *Lucid lacks features needed by real programmers*

Basic Lucid as described in [3] is a very spartan language, there being no procedures, no data structures (in particular no arrays), no control structures and no I/O.

---

* The first value of *A* is, by convention, the value of *A* at time 0.

a) Procedures literally make no sense in Lucid, but functions *do* make sense [5]. Procedures can be simulated by using functions which return several values, just as blocks are simply phrases (compound expressions) which return several values. Functions, like phrases, can refer to global variables, but cannot change them (that wouldn't make sense semantically). In other words, there are automatically no side-effects.

b) Control structures make no sense in Lucid. This does not mean that the Lucid user has no control over execution of his or her program. If a particular implementation uses various operational devices to run programs then the user can, to a certain extent, determine the behaviour of the program by altering the form of the program (for example, putting it into a form with a simple iterative interpretation). The user does not need precise control of the program's behaviour because the semantics of the program is independent of operational considerations. The Lucid user specifies the meaning of the program exactly but only suggests (by the form of the program) its behaviour. This presupposes an implementation which selects the *method* of executing the program by analysing the form of the program. The mathematical nature of Lucid (lack of side effects, etc.) makes such analysis feasible.

Incidentally, some "control structures" have, in reality, nothing to do with control and *can* be used in Lucid without any problems (e.g. conditional expressions, case expressions).

c) Data structures do make sense in Lucid, because Lucid was defined independently of data objects. If you want lists, for example, you just have to base your version of Lucid on an algebra of lists and operations on them. (These operations must, of course, be mathematical functions, i.e. without side-effects.) For arrays, the operations could include an "update" operation U_n, for each number of dimensions n, such that $U_n(A,i_1,i_2,i_n,k)$ is the array similar to the n-dimensional array A except that its $i_1,i_2,\ldots,i_n$-th component is k. In this way, Lucid could "handle" the normal way of using arrays in algorithms, where arrays are changed incrementally. Probably a better way to use arrays is to use the APL approach of having operations which take whole arrays as arguments and return whole arrays as results. This fits in better with the Lucid philosophy of specifying the results of computations rather than the details of how these results are to be achieved. Also, the programs tend to be simpler and more understandable.

An alternative approach to arrays, which requires some extension of the semantics of Lucid, is to have functions which convert histories into one-dimensional arrays, and vice versa. These arrays are naturally infinite, and we call them "chains". Chains can only be processed linearly, and are not "random access", but are very natural to use for certain problems.

Again there is no explicit control over the operational behaviour of programs, but implicit control is possible.

d) Lucid cannot have commands for inputting and outputting data. A Lucid program is a term whose value is taken as the output of the program, and whose free variables are the input variables. If the input

and output are streams, an implementation can be devised in which input is demanded when needed and output is produced when available. If the input to the program depends, via the user, on previous outputs, a 'dialogue' with the program is set up.

e) Another feature missing in Lucid is error exits. Errors can instead be handled by suitable use of "error objects", like "integer overflow", "subscript range error" etc., which are the result of operations applied to operands not in their normal domain. The basic data algebra of the language must be extended to allow such objects.

f) Since there are no control structures, there is of course no goto statement. Several languages nowadays do not have such a statement, but this was always the result of a conscious design decision - transfers make sense but are not permitted. In Lucid, goto statements would be meaningless, since there is no concept of 'execution' being at any particular 'point' in the program anyway. Like side-effects, the question of allowing them or not does not arise. They just have no place in Lucid.

The main features lacking in Lucid are control structures, and a little practice in programming in Lucid will reveal this lack to be a positive asset. Not having to worry about control flow is remarkably liberating.

4. *Lucid is inherently inefficient*

This is probably the most common misconception. It arises naturally since most other languages have essentially one implementation, determined by the semantics. Since the Lucid semantics of even the simplest program involves infinite objects, the conclusion appears inescapable.

The flaw in this argument is that the Lucid semantics, being mathematical, specifies goals (or ends) of an implementation, not the means. For example, the computation of the value of the program

```
valof
    X = 0 fby X+1
    Y = 1 fby Y+2·X+3
    result = X asa Y>N
end
```

at (external) time t when N has the value 10 does not require the computation of an infinite number of values of the variables X and Y. It requires the value of *result* at (local) time t. This requires only the values 0,1,2 and 3 and 1,4,9 and 16 of X and Y respectively at local times 0,1,2 and 3, because at this last stage $Y > 10$ is true for the first time and the corresponding value of X, namely 3, is the value of X asa $Y > 10$ at any time, in particular at time t. Thus the value of the program at time t is 3, the integer square root of 10.

This program has been especially simple, but even for more complicated programs that don't correspond clearly to iterative programs it will always be the case that if the value of the program at a particular time is defined, according to the mathematical semantics, then only a finite number of the values of the variables need be calculated in order to determine what this defined value is.

There is in fact an implementation based directly on the mathematical semantics which computes *only* the values which are required. This is a demand-driven interpreter [6] where a demand to compute a particular variable at a given time generates other demands for other variables at other times. In its simplest form it is inefficient because if the value of a variable at a given time is needed again it will be recomputed from scratch. It can be made more efficient by saving some of the computed values, but once again it is inefficient, in space this time, if *all* the values are saved. Finding an appropriate balance is one of the main problems of implementing Lucid using an interpreter.

This implementation scheme can be used when variables represent arrays, but it can then be inefficient if the recommended APL-like style of programming is not adopted, and the incremental approach, using U_n, is used instead. In this case the implementation will be keeping around several copies of arrays, differing in slight ways. It is possible, with some analytical overhead, to avoid copying an array, with some updating, by keeping just the updating *information* around until the previous array is not needed any more, at which point the actual updating can be performed. Then instead of several copies of an array we will have one copy together with information on various updates to be performed.

In general it will not be the case that we can put a bound on the number of different values of a variable (or the number of updates) that need to be remembered. It is quite possible to write programs that will be enormously expensive to compute. (This is true for other languages also.) However, it appears that many programs (like the one given) have the property that a good interpreter would only need a bounded number of instances (for different times) of each variable. In such cases, the program can be executed in a different way - it can be translated into a conventional iterative program, and this program can be executed. In other words, such programs can be compiled rather than interpreted [7,8].

It is clear that the fact that the semantics of Lucid is mathematical rather than operational leaves a great deal of freedom to the implementer to come up with ingenious, efficient implementations.

5. *Practically no Lucid programs can be implemented as iterative programs, data-flow nets or coroutines*

This point of view contrasts sharply with that considered in section 2. The crux of the problem here is seen to be that the Lucid semantics is mathematical and, in order for a program to have a defined value at a particular time, this value depends only on the values of

certain variables at certain times and is independent of the values of all other variable/time combinations, even undefined values. The value of a program is independent of the results of irrelevant computations. Any implementation which attempts to evaluate a particular variable at a particular time, without first being certain that that value is definitely needed for the value of the program, runs the risk of getting stuck in a non-terminating computation when, in fact, the value of the program *is* defined, according to the semantics. In Lucid it is meaningful for a sequence or stream or history to be "intermittent", that is, it can contain undefined values ($\perp$) followed by defined ones, and these later defined ones may be crucial for determining the value of the program.

In a "pure data-flow" implementation of Lucid [18] the sequences of data items flowing along the data lines should correspond exactly to the histories of streams in Lucid. This is not possible if the streams are intermittent since this would require the recognition of non-terminating computations in order to produce an object corresponding to $\perp$.

This isn't as unfortunate as it seems, because pure data flow is too restrictive to implement even conventional programming languages. For example, in order to implement conditional expressions, pure data flow uses a 3-input node, these inputs corresponding to the test and the two values corresponding to the two branches of the conditional. When the node has received the value of the test and the value for the appropriate branch, this latter value is passed through. However, it is not possible to subsequently select the other branch until the value of that branch, corresponding to the value that was passed through, has arrived and been discarded. Even in cases where this produces the right answer (i.e. it doesn't cause the program to 'hang up'), it performs more computation than necessary in order to do so.

Pure data flow has to be modified to avoid doing unnecessary computations, and this is exactly what is necessary in order to correctly implement Lucid. It seems that by using sophisticated analysis techniques, data-flow nets can be constructed for the programs in a large subset of Lucid.

Similar problems occur with simple coroutine implementations and simple compilers into iterative programs. However, using more sophisticated algorithms, both these techniques can be extended to large subsets of Lucid.

Similar problems occur with simple coroutine implementations and simple compilers into iterative programs. However, using more sophisticated algorithms, both these techniques can be extended to large subsets of Lucid.

Some people may be tempted to tinker with the semantics of Lucid in order to avoid things like intermittent streams, but the result would be a language that didn't have many of the nice properties of Lucid. Anyway, since pure data flow is inadequate anyway there doesn't seem to be much point in getting a language that can be implemented in this way.

6. *Lucid is too strange for ordinary programmers to use*

On first sight, Lucid does have many strange properties:

i) order of statements within phrases is irrelevant;
ii) the values of variables are infinite sequences;
iii) statements are just equations;
iv) there is no flow of control, no idea of where the execution of the program is 'at'.

The usual conclusion is that it is difficult to write Lucid programs. This is certainly true for many people, but this is not because Lucid is unreasonably weird but because it really requires a completely different style of programming.

The question that should be asked is not which style is strange compared to which, but rather which style is better. Programmers used to the imperative style of programming usually see Lucid as a very restrictive language but (as indicated earlier) the exact opposite is true.

For one thing, the Lucid style is freer because it is not committed to one particular operational viewpoint. An equation like

$$I = 1 \text{ fby } I+1$$

can be understood

a) as defining successive values of a loop variable
b) as defining an infinite sequence
c) as defining a module which spontaneously generates the numbers 1,2,3,...

or

d) as a coroutine which supplies the numbers on demand.

Some of these meanings allow for a more modular view of programming than is found in imperative programming. For example, we can understand this first equation by itself and then add the equation

$$P = \text{first } X \text{ fby } P \cdot \text{next } X \ .$$

P gives us the running products of the values of X, that is, the factorials of the successive positive integers. In a conventional language we would have various assignments to P and X intermingled in a loop, and the order of these assignments could be crucial. It would be less obvious that the roles of P and X can be understood separately, and the program would be harder to modify.

Above all, in Lucid the programmer is freed from having to understand (and specify) exactly what is going on.

We have also seen how functions in Lucid can be thought of as coroutines, and in fact Lucid programs using functions are much clearer and easier to write than programs in existing coroutine languages.

Tongue-in-cheek comment. It is quite possible that programmers of the future will look back at imperative languages and find them strange, and wonder at the fact that programmers were able to write programs in them, in much the same way that we wonder at the ancients' ability to do arithmetic with Roman numerals. *(End of tongue-in-cheek comment.)*

7. *Lucid has no limitations*

In spite of all we have said, we are forced to admit that there are genuine limitations in the use of Lucid as a programming language. We don't consider this to be a failure because we never intended for Lucid to be able to "handle" everything.

One very fundamental property of Lucid programs is that they are definitional. If you want the value of a variable to be an integer between 1 and 10 you have to specify exactly which one you want. If you are going to use a Lucid loop to calculate the sum of some numbers, you have to specify the way in which the numbers are to be combined. Obviously these are examples in which the programmer is required to over-specify.

At first sight, this would seem to be an easy problem to fix, by simply adding McCarthy's *amb* function [14]; *amb*(a,b) arbitrarily returns *a* or *b*, and we can't predict which. Unfortunately this destroys the mathematical semantics, the rules of inference, and consequently the whole of Lucid.

There may be another way to weaken the specifications, and that is to make them general assertions rather than equations. This means basing the language on predicate calculus rather than USWIM [4] (which is a variant of ISWIM, which is based on λ-calculus). However, to develop such a genuinely assertional programming language is a large project which we must leave to the future.

Lucid's main strength as a programming language is that no behaviour is specified, but this is also a weakness when control of behaviour is the *aim* of a program, as in real-time computing. We talk about values at certain "times" for interpreters, but these are not *real* times. To control behaviour we must consider particular implementations or else modify the formal semantics to take account of real time, using possibly a "clock pulse object" (which the second author has named a "hiaton").

The fact that Lucid has limitations does not signify failure. Trying to obtain too much generality in a single language has in the past often led to failure. Moreover, although the particular language, Lucid, is a product of the general philosophy of using mathematics as the basis of language design, the fact that Lucid has some limitations should not be taken as implying that there are limitations to the mathematical approach. Lucid is only one language that was designed this way. The mathematical approach has also been used, to varying extents, to design LISP, ISWIM, APL and PROLOG, and we certainly expect other new languages to be designed this way, especially since there are now many powerful new results in the theory of domains [15] that can be used.

REFERENCES

1. Arsac, J., "La Construction de programmes structurés", Dunod, Paris (1977).

2. Arvind, Gostelow, K.P. and Plouffe, W., "The (Preliminary) Id Report", Department of Information and Computer Science (TR114a), University of California at Irvine (1978).

3. Ashcroft, E.A. and Wadge, W.W., "Lucid, a Nonprocedural Language with Iteration", CACM, 20, No. 7 (1977).

4. Ashcroft, E.A. and Wadge, W.W., "A Logical Programming Language", CS-79-20, Computer Science Department, University of Waterloo (1979).

5. Ashcroft, E.A. and Wadge, W.W., "Structured Lucid", CS-79-21, Computer Science Department, University of Waterloo (1979).

6. Cargill, T.A., "Deterministic Operational Semantics of Lucid" CS-76-19, Computer Science Department, University of Waterloo (1976).

7. Farah, M., "Correct Translation of Lucid Programs", in Program Transformations, Proceedings of the 3rd International Symposium on Programming, Dunod, pp. 367-380 (1978).

8. Hoffman, C.M. "Design and Correctness of a Compiler for a Nonprocedural Language", Acta Information 9.

9. Kahn, G. and McQueen, D.B., "Coroutines and Networks of Parallel Processes", IFIP 77, pp. 993-998 (1977).

10. Kleene, S.C., "Introduction to Meta Mathematics", Van Nostrand, Princeton (1952).

11. Landin, P.J., "The Next 700 Programming Languages", CACM 9, No. 3 pp. 157-166 (1966).

12. May, M.D., Taylor, R.J.B. and Whitby-Strevens, C., "EPL: an Experimental Language for Distributed Computing", Proceedings of "Trends and Applications: Distributed Processing", National Bureau of Standards, pp. 69-71 (May 1978).

13. McCarthy, J. et al., "LISP 1.5 Programmer's Manual", M.I.T. Press (1962).

14. McCarthy, J. "A Basis for a Mathematical Theory of Computation", in Computer Programming and Formal Methods, North Holland, Amsterdam pp. 33-70 (1963).

15. Scott, D., "Data Types as Lattices", SIAM J. on Comput 5, No. 3, pp. 522-587 (1976).

16. Stoy, J.E., "Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory", M.I.T. Press (1977).

17. Tesler, L.G. and Enea, H.J., "A Language Design for Concurrent Processes", Spring Joint Computer Conference, pp. 403-408 (1968).

18. Wadge, W.W. "An Extensional Treatment of Dataflow Deadlock", Lecture Notes in Computer Science, No. 70, Springer Verlag (1979).