DEMYSTIFYING PROGRAM PROVING: AN INFORMAL INTRODUCTION TO LUCID

by
E.A. Ashcroft
and
W.W. Wadge*

Research Report CS-75-02
June 1975

Computer Science Department
University of Waterloo
Waterloo, Ontario, Canada

* Computer Science Department, University of Warwick, England

0. Introduction

There has been much work done recently on techniques of program proving, but nevertheless most programmers still make little if any effort to verify their programs formally. Perhaps the main obstacle is the fact that most programming languages are not 'mathematical' despite their use of some mathematical notation. This means that in proving a program it is necessary either to translate the program into mathematical notation (e.g. into the relational calculus) or to treat the program as a static object to which mathematical assertions are attached. In either case, the language in which assertions and proofs are expressed is different (often radically different) from the language in which programs are written. Moreover, wildly nonmathematical features such as pointer variables and side effects make it even more difficult for proofs to be completely formal.

Our aim is to overcome this obstacle with a single formal system called Lucid in which programs can be written and proofs carried out. A Lucid program can be thought of as a collection of commands describing an algorithm in terms of assignments and loops; but at the same time Lucid is a strictly denotational language, and the statements of a Lucid program can be interpreted as true mathematical assertions about the results and effects of the program. For example, an assignment statement in Lucid can be considered as a statement of identity, an equation. A correctness proof of a Lucid program proceeds directly from the program text, the statements of the program being the axioms from which the properties of the program are derived, the rules of inference being basically those of first order logic with quantifiers. Furthermore, in Lucid we are not restricted to proving only partial correctness or only termination or only equivalence of programs - Lucid can be used to express many types of reasoning.

Formal details of the syntax and semantics of Lucid, and of the rules of inference and their formal justification are given in [1]. In this paper we will describe the language and rules of inference informally, outline a sample proof and also indicate ways of implementing the language. (Several implementations have been completed, and others are under development.)

1. General Principles

There already exist formal mathematical systems which can serve as programming languages. For example, the following recursion equations

root(n) = s(0,1,n)

s(i,j,n) = if j>n then i else s(i+1,j+2i+3,n)

can be considered both as a recipe for computing the integer square root of n and also as assertions about the partial functions root and s. From the program, considered as a set of assertions, we can formally derive the assertion

$$root(n)^2 \leq n \;\&\; n < (root(n)+1)^2.$$

The problem is that most programmers find the purely recursive approach too restrictive and are therefore likely to use iteration to express the same square root algorithm, in an "imperative" language such as FORTRAN:

```
      INTEGER I,J
      READ,N
      I = 0
      J = 1
10    IF(N.GT.N) GO TO 20
      J = J + 2*I + 3
      I = I + 1
      GO TO 10
20    WRITE,I
      STOP
```

Although statements like I = 0 are suggestive of mathematics, the program as a whole cannot be considered as a set of assertions because of statements such as I = I+1 and GO TO 10, which are nonsense or meaningless as mathematics.

The two main nonmathematical features in programming languages are transfer and assignment, but the difficulty in eliminating them is the fact that iteration seems to make essential use of these features, and programmers find them 'natural'. So if we are to keep iteration we must find a way of making assignment and control flow mathematically respectable.

Lucid does this by explicitly distinguishing between the initial value of a variable in a loop (first I), the value of the variable during the current iteration (simply I) and the value on the next iteration (next I). In addition, Lucid has the binary operation as soon as which extracts values from a loop. In Lucid the square root program is

(1) N = first input

(2) first I = 0

(3) first J = 1

(4) next J = J + 2×I + 3

(5) next I = I + 1

(6) output = I as soon as J>N

The meaning of the program, considered as a collection of commands, is fairly clear: statement (1) inputs N, statements (2) and (3) initialize the loop variables I and J, statements (4) and (5), when executed repeatedly, generate successive values of the loop variables, and statement (6) terminates the loop and outputs the result. The loop is implicit in the use of first and next.

In more detail, the statement "first I = 0" asserts that the initial value of I is (was) 0; the statement "next J = J + 2×I + 3" asserts that at

each stage in the iteration the value of J at the next stage is the current value of J plus twice the current value of I plus three; and the statement "output = I as soon as J>N" asserts that the output is the value of I when J is for the first time greater than N. Notice that the order of the statements is irrelevant, in particular it does not matter if we reverse (4) and (5).

But we can also consider the statements as true mathematical assertions about the histories of the variables. More precisely, we define a history (or even better, a "world-line") to be an infinite sequence, i.e. a function with domain N ($=\{0,1,\ldots\}$). Then variables and expressions in Lucid formally denote not single data objects, but rather infinite sequences of data objects. The i-th value of the sequence which, say, "X" denotes can be thought of as the value which X would have on the i-th iteration of the loop for X, if the program in which X occurs were executed as a set of commands.

If the variables and expressions are to formally denote sequences, then symbols like "+" and "next" must be interpreted as denoting operations on sequences. The ordinary data operations and relations and logical connectives work 'pointwise': if "X" denotes $\langle x_0,x_1,x_2,\ldots\rangle$ and "Y" denotes $\langle y_0,y_1,y_2,\ldots\rangle$ then "X+Y" must denote $\langle x_0+y_0,x_1+y_1,x_2+y_2,\ldots\rangle$, because the value of X+Y on the i-th iteration will be the value of X on the i-th iteration plus the value of Y on the i-th iteration. Note that "3", for example, must denote the infinite sequence each component of which is three. Note also that the 'truth value' of an assertion such as "X≥Y" is also an infinite sequence and so may be neither "T" (each component true) nor "F" (each component false).

The meaning of the special Lucid functions (first, next, etc.) can now be made clear. The value of next X on the i-th iteration is the

value of X on the i+1st iteration; thus if "X" denotes $\langle x_0,x_1,x_2,\ldots\rangle$, "next X" denotes the sequence $\langle x_1,x_2,x_3,\ldots\rangle$; similarly "first X" denotes the sequence $\langle x_0,x_0,x_0,\ldots\rangle$. Furthermore, if "P" denotes $\langle p_0,p_1,p_2,\ldots\rangle$ then "X as soon as P" denotes $\langle x_j,x_j,x_j,\ldots\rangle$, where j is the unique natural number such that p_j is true and p_i is false for all i less than j (X as soon as P is undefined if no such j exists, i.e. it is an infinite sequence of undefined elements). (Later we will also use a function hitherto, where "hitherto P" denotes $\langle true,p_0,p_0 \,\&\, p_1,p_0 \,\&\, p_1 \,\&\, p_2,\ldots\rangle$.)

Applying these definitions to the statements of the Lucid square root program, we see that statements (2)-(5) can be true only if I is $\langle 0,1,2,\ldots\rangle$ and J is $\langle 1,4,9,16,\ldots\rangle$. Furthermore, if input is, say, $\langle 12,8,14,\ldots\rangle$ then N must be $\langle 12,12,12,\ldots\rangle$, J > N must be $\langle false,false,false,true,\ldots\rangle$ and so output, which is equal to I as soon as J > N, must be $\langle 3,3,3,\ldots\rangle$. This result agrees with our intuitive understanding of the effect of 'executing' the program in the conventional sense.

Using this semantics we can derive general axioms and rules such as

first(X+Y) = first X + first Y

which allow us to reason about programs without referring explicitly to sequences. In fact, proofs can be made knowing very little of the formal semantics. It is easily verified that all the basic rules of inference of first order logic (e.g. from A and B infer A & B) are valid, with two exceptions: the law of the excluded middle and the deduction theorem. The first rule, the law of the excluded middle, asserts that either A or ¬A is true,(¬A means "not A") and it fails because A may be undefined (the 'result' of a computation that does not terminate). The second rule, the deduction theorem, allows us to

infer A → B ("A implies B") from a proof of B which has A as an assumption (this "discharges" the assumption). It fails because of the way in which the truth of Lucid assertions depends on time. For example, from I = 0 ("I is always zero") we can derive next I = 0, but I = 0 → next I = 0 is not valid because → and = work pointwise. This type of inference is correct, however, if in the proof of B we did not substitute equals for equals within the scope of a Lucid function, and did not use any special rule, like the induction rule, which depends on Lucid functions. (The induction rule is given below.) This restricted sort of reasoning corresponds to considering one particular stage in an iteration.

2. A Sample Proof

Our goal is to derive the assertion $\text{output}^2 \leq \text{first input} < (\text{output}+1)^2$ from the text of the Lucid program (considered as a set of assertions) together with the assumption integer first input & first input ≥ 0.

The first step is to establish $J = (I+1)^2$ using the basic Lucid induction rule, which states that for any assertion P,

first P, P → next P |= P

(where for any assertion A and set Γ of assertions, Γ |= A means that if everything in Γ is true, then A is true).

Taking P to be "$J = (I+1)^2$" we have

$$\begin{aligned} \text{first } P &= \text{first}(J = (I+1)^2) \\ &= (\text{first } J = (\text{first } I+1)^2) \\ &= (1 = (0+1)^2) \end{aligned}$$

which, of course, is true. Now we assume that $J = (I+1)^2$ is true at some stage in the iteration. Then we have

$$\begin{aligned} \underset{\sim}{\text{next}}\ J &= J + 2\times I + 3 \\ &= (I+1)^2 + 2\times I + 3 \quad \text{(by the assumption)} \\ &= ((I+1) + 1)^2 \\ &= (\underset{\sim}{\text{next}}\ I+1)^2 \\ &= \underset{\sim}{\text{next}}((I+1)^2) \end{aligned}$$

and so we have $\underset{\sim}{\text{next}}(J = (I+1)^2)$. Thus

$$(J = (I+1)^2) \rightarrow \underset{\sim}{\text{next}}(J = (I+1)^2)$$

is always true, since we were reasoning only about a single stage in the iteration (and consequently didn't substitute within the scope of a Lucid function or use "time dependent" rules). We can now apply the induction rule and conclude that $J = (I+1)^2$ is always true.

(Reasoning using the induction rule is the Lucid analog of the inductive assertion method of program proving. Note that we use properties of integers in the proof in a very cavalier manner, without worrying that we are actually talking about infinite sequences of integers. This is one of the beauties of Lucid proofs.)

In Lucid, "=" denotes true equality so since we have proved that $J = (I+1)^2$ we may infer that J and $(I+1)^2$ are identical (have the same histories) and so every occurrence of J in the program may be replaced by $(I+1)^2$. This produces the following equivalent "cleaned up" program:

```
N = first input
first I = 1
next I = I+1
output = I as soon as (I+1)^2 > N
```

To complete the proof we use properties of the function as soon as. The first is an axiom stating that the result of the function is constant:

first(X as soon as P) = X as soon as P.

The second is a rule which appears rather complicated. Let ψ be an assertion containing a free variable X, and ψ' be the result of replacing all free occurrences of X in ψ by X as soon as P. The rule is then

$$P \ \&\ \underline{\text{hitherto}}\ \neg P \rightarrow \psi,\ \underline{\text{first}}\ \psi' = \psi',\ \underline{\text{eventually}}\ P \models \psi'.$$

Informally, this says that if assertion ψ is true of X when P is true for the first time, and ψ is constant apart from X, then provided P eventually does become true we know that ψ is always true of X as soon as P. (To prove first $\psi' = \psi'$ we must use the axiom above. The expression eventually P is just shorthand for T as soon as P.)

In this example we apply the rule by first establishing by induction that

$$\underline{\text{hitherto}}\ ((I+1)^2 \leq N) \rightarrow I^2 \leq N.$$

This is straightforward using first input ≥ 0 and the facts that

first hitherto P = T

and next hitherto P = P & hitherto P.

We thus have a suitable premise for the as soon as rule:

$$(I+1)^2 > N\ \&\ \underline{\text{hitherto}}((I+1)^2 \leq N) \rightarrow I^2 \leq N < (I+1)^2.$$

So here ψ is $I^2 \leq N < (I+1)^2$, and P is $(I+1)^2 > N$. Note that ψ is constant apart from I, or more formally

$$\begin{aligned}
\underline{\text{first}}\ \psi' &= \underline{\text{first}}((I \text{ as soon as } P)^2 \leq N < ((I \text{ as soon as } P) + 1)^2) \\
&= (\underline{\text{first}}(I \text{ as soon as } P))^2 \leq \underline{\text{first}}\ N < (\underline{\text{first}}(I \text{ as soon as } P) + 1)^2 \\
&= (I \text{ as soon as } P)^2 \leq N < ((I \text{ as soon as } P) + 1)^2 \\
&= \psi'.
\end{aligned}$$

This gives us the second premise for the as soon as rule.

We can establish the third premise, eventually P, by using the following 'termination rule':

integer A, A > next A |= eventually(A ≤ 0).

To use this, we take $N+1-(I+1)^2$ for A, which is clearly integer, and we have

$$N+1-(I+1)^2 > \text{next}(N+1-(I+1)^2),$$

because this reduces to $(I+1)^2 < (I+2)^2$, and we know that I is positive. Thus we have that eventually($(I+1)^2 \geq N+1$), i.e. eventually($(I+1)^2 > N$).

We now apply the as soon as rule, giving ψ', and since output = I as soon as $(I+1)^2 > N$ and N = first input, ψ' is the same as $\text{output}^2 \leq$ first input $< (\text{output}+1)^2$, which is the correctness condition. Note that this implies that output is defined; proving eventually($(I+1)^2 > N$) is the Lucid analog of proving termination for this program.

3. Proofs

The example has illustrated most of the axioms and rules for simple Lucid proofs. We will summarise these here. The formal system is given in detail in [1].

Lucid proofs proceed mainly by straightforward mathematical reasoning, using properties of the data domain, and properties of logic. In addition, we use properties of the Lucid functions.

Properties of the Data Domain

Any axioms or rules of inference that are valid for the basic data domain D are also valid in the context of Lucid proofs. For example, in the proof above we used axioms like

$0+1 = 1$ and

$\forall x[(x+2)^2 = x^2 + 4\times x + 4]$.

However, there is one thing to be careful of. The data domain D must include an 'undefined' element $\perp$, and the axioms and rules must be valid in the presence of this undefined element. For example, $\forall x\ \neg(x = x+1)$ is <u>not</u> valid since $\perp = \perp+1$.

Logical Properties

In the same way we can use most of the axioms and rules in inference of ordinary logic. The few we cannot use fail either because we must allow an 'undefined' truth value, or because we are talking about sequences. As explained earlier, the law of the excluded middle fails because of the undefined truth value, and the deduction theorem fails because of sequences.

Since the law of the excluded middle fails so does reasoning by contradiction. Also certain propositional calculus tautologies become invalid, such as $(A \rightarrow B) \rightarrow \neg A \vee B$.

There are weaker versions of the law of the excluded middle, and reasoning by contradiction, that <u>do</u> work, namely, $(A = T) \vee \neg(A = T)$ is true, and from $A \rightarrow F$ we can conclude $\neg(A = T)$.

With practice it is easy to avoid the few pitfalls caused by the undefined truth value.

We can also regain the deduction theorem, as explained previously, by reasoning essentially about a single stage in a computation. This turns out to mean that we cannot substitute equals for equals within the scope of a Lucid function, or use any of the Lucid rules given in the next section. (Lucid axioms are admissible.)

Lucid Properties

The following lists are not exhaustive.

Axioms

(a) The most useful property is that first and next commute with data operations and relations and logical connectives,
e.g. first A+B = first A + first B

(b) first first X = first X & next first X = first X

(c) first(X as soon as P) = X as soon as P &
next(X as soon as P) = X as soon as P

(d) first hitherto P = T &
next hitherto P = P & hitherto P

Rules of inference

(a) P |= first P

(b) P |= next P

(c) first P, P → next P |= P

(d) P & hitherto ¬P → ψ(X), first ψ(X as soon as P) = ψ(X as soon as P), eventually P |= ψ(X as soon as P)

(e) integer A, A > next A |= eventually (A ≤ 0).

4. Programming

This paper is not intended to be a guide to programming in Lucid, but to indicate the new approach to program proving that the ideas behind Lucid provide. In fact, Lucid itself is being developed and extended in various ways,

and to publish a programming manual now would be premature.

However, it is important to realise that Lucid manages to treat assignment statements as equations, and to make loops implicit, only by making drastic restrictions on the control structure of programs and on the ways in which assignment statements can be used. We will consider these restrictions in this section.

Every Lucid program can be considered as being built up from nested and concatenated loops. Every variable X which is "defined" by first X = $\mathcal{E}_1$, next X = $\mathcal{E}_2$, for expressions $\mathcal{E}_1$ and $\mathcal{E}_2$, implies the existence of a loop, and X is called a 'loop-variable'. These basic loops can be linked together in various ways.

Two such loops can be synchronised, giving essentially one loop, if one refers to the other. For example, if first X = 1, next X = X+1, first Y = 1 next Y = Y+X+1, the loops for X and Y are synchronised because to compute the values of Y for the next iteration we need the 'current' values of both X and Y.

We can extract particular values from a loop using as soon as, as we did in the square root program. This gives another way of linking loops, by extracting a value from one loop, and using it in another (or in an as soon as test of the other). This implies that the first loop has to be computed first, and the loops are concatenated. For example, if

```
first Y = 1
next Y = Y+Y
X = Y as soon as Y > N
first Z = X+1
next Z = Z+1
```

the Z loop cannot start until a value for X has been picked out of the Y loop. In the following example

```
first Y = 0
next Y = Y+1
first S = Y×Y
next S = S + Y×Y
N = S as soon as Y eq M
first I = 0
next I = I+1
first J = 1
next J = J + 2×I + 3
output = I as soon as J>N
```

the Y and S loops are synchronised, and the I and J loops are synchronised. The Y,S and I,J loops are concatenated, in an indirect way; values for output cannot be picked out of the I,J loop until a value is known for N, which means the Y,S loop must go first. (This program sums the squares of the integers 0 through M, and outputs the integer square root. The relation 'eq' is 'computable equality'; its value is undefined if either of its arguments are undefined.)

Note in the above example that Y = I. We can therefore simplify the program by eliminating Y:

```
first I = 0
next I = I+1
first S = I×I
next S = S + I×I
N = S as soon as I eq M
first J = 1
next J = J + 2×I + 3
output = I as soon as J>N.
```

Here the I and S loops are synchronised, and the I and J loops are synchronised, yet the S and J loops are concatenated. This example illustrates that it is possible to write programs whose control structure is difficult to interpret in terms of conventional concepts. However, by adding copies of variables, like Y for I, it is possible to get to equivalent programs with straightforward control structure.

We can also have one loop nested within another, as will be shown in the next section. With these ways of connecting loops together we have the basic control structures of structured programming, except for the conditional statement. Instead of the conditional statement, in Lucid we use the conditional expression. (Semantically, the ternary function if ... then ... else is just another function, like addition for example, that works pointwise on histories.) Thus we can write Euler's algorithm for finding the gcd of two integers N and M as follows:

```
first X = N
first Y = M
next X = if X>Y then X-Y else X
next Y = if X>Y then Y else Y-X
output = X as soon as X eq Y
```

(We can extend the language to avoid duplication of tests by using vectors of variables. Thus the two statements with conditional expressions could be replaced by the following single statement

```
next (X,Y) = if X>Y then (X-Y,Y) else (X,Y-X).)
```

This example also illustrates one of the restrictions on assignments. Each loop variable must be updated each time around the loop, even if

its value is not changed. Moreover, a variable can be updated only once on each iteration; if some intermediate value is needed it must be held in a separate variable.

The main restriction on assignment is that each variable can only be assigned to at one place in the program (with loop variables there can be one initialisation and one updating assignment). There are other restrictions on assignments when we consider nested loops.

5. Nested Loops

A completely general approach to iteration must allow nesting of loops. For example, to test a positive integer N for primeness we might generate successive values 2,3,4,... of potential non-trivial divisors of N, using a variable I say, and for each value of I use an inner loop to generate multiples of I, using a variable J. (It is sufficient to generate multiples of I starting with I^2.) Some value of J will be equal to N if and only if N is not prime.

In Lucid the program Prime is as follows:

```
N = first input
first I = 2
begin
   first J = I×I
   next J = J+I
   IdivN = J eq N as soon as J≥N
end
next I = I+1
output = ¬IdivN as soon as IdivN ∨ I×I≥N.
```

The inner loop is delimited by begin and end. Intuitively the inner loop is invoked on each iteration of the outer loop. Within the inner

loop, the 'global' variable I is constant (I = first I) for each invokation, fixed at the value of I for the current iteration of the outer loop. Each invokation of the inner loop produces a constant truth value for the variable IdivN, which is then used in the current iteration of the outer loop. The variables I, IdivN and N are called globals of the inner loop, because they are used outside the loop. Inside a loop, globals are constant, though they may vary outside the loop (here both I and IdivN are not constant in the outer loop).

We see that globals like I mean different things inside and outside loops. Outside the inner loop we have next I = I+1 but inside the loop next I = I, since I is constant. Clearly, we must take care, in proofs, to keep track of the 'level' at which something is true. Both these assertions cannot be true at the same level (either both in the inner loop, or both in the outer loop) or we would have I = I+1. However, if we do take care of the levels in this way, we don't get into trouble.

There are some assertions which are true at more than one level. In fact, any assertion without Lucid functions that is true at some level is true at all lower levels (within inner loops) and if in addition it refers only to globals of this level, it is also true one level higher (in the enclosing loop). For example, in the Prime program, we can prove I>1 at the outer level, and this is then also true within the inner loop. (We say we can 'move the assertion' into the inner loop.) Using this, and I = first I, we can then prove J>I within the inner loop, but this assertion can not be moved to the outer level, because it refers to J, which is not a global. However, we can prove

$$\text{IdivN} = \exists K\ I \le K < N \ \&\ I \times K = N$$

in the inner loop, and this can be moved out. In fact, continuing the proof, provided first input > 0, we can actually get

output = $\neg\exists L\ \exists K\ 2 \le K \ \&\ K < \underset{\sim}{\text{first}}$ input $\ \&\ L \times K = \underset{\sim}{\text{first}}$ input

at the outer level (see [2]), which proves that the program is correct.

To prove things about programs with nested loops we just have to make 'nested proofs'.

These 'meta-rules' for doing nested proofs are justified by the following formal semantics of nested loop programs. Without nesting, the history of a variable is simply a sequence, a function from the natural numbers to D, the set of data objects (including 'undefined'). With nesting, histories are more complicated, and in general depend on more than one time parameter.

Each variable or constant occurrence has a level - the number of loops within which it occurs. (With no nesting, everything has level 1.) The level is also the number of time parameters on which the value denoted by this variable or constant occurrence depends. Thus I in the outer loop depends on one time parameter, the number of iterations of the outer loop. In the inner loop I and J depend on two time parameters, the numbers of iterations of the inner and outer loops. I_{t_0} is the value at 'time' t_0 of level 1 occurrences of I, and $I_{t_0} = t_0 + 2$. $I_{t_0t_1}$, $J_{t_0t_1}$ are the values at 'time' t_0t_1 of level 2 occurrences of I and J, and we have $I_{t_0t_1} = t_1 + 2$, and $J_{t_0t_1} = (t_1+2)(t_0+t_1+2)$. Note that $I_{t_0t_1}$ is independent of t_0, since level 2 occurrences of I are constant.

In $A_{t_0t_1t_2\ldots}$, t_0 is the number of iterations at the level of the occurrence of A, t_1 is the number of iterations of the enclosing loop, t_2 is the number of iterations of the loop enclosing that one, and so on. Then all the functions, including Lucid functions, are defined as pointwise extensions to the 2nd, 3rd etc. time parameters of the level 1 functions we used when there was no nesting

For example, $(\underset{\sim}{\text{next}}\ A)_{t_0t_1t_2} = A_{t_0+1t_1t_2}$. Note that with these definitions the values of $I_{t_0t_1}$ and $J_{t_0t_1}$ stated above satisfy next J = J+1, as required. Also it is easy to see that the meta-rules for nested proofs are valid.

Although adequate, this definition of the semantics is inelegant in some respects. We can avoid the distinctions between different level occurrences of the same variable by introducing a function latest, which makes the begin...end notation redundant. The function latest increases the number of time parameters: $(\underset{\sim}{\text{latest}}\ A)_{t_0t_1..} = A_{t_1..}$. Thus the program Prime can be written

N = first input

first I = 2

first J = latest I×latest I

next J = J+latest I

latest IdivN = J eq latest N as soon as J≥latest N

next I = I+1

output = ¬IdivN as soon as IdivN ∨ I×I≥N

and for all occurrences of I, $I_{t_0} = t_0+2$. (The rule for removing begin and end is to replace all contained occurrences of a global variable X by latest X. To be strictly correct we should also do the same for constants, but this becomes unnecessary with the next improvement.)

Having got rid of begin and end we lose the concept of level. It is then inelegant that different variables can depend on different numbers of time parameters. For uniformity we should rather consider all variables and constants to depend on an infinite number of time parameters, probably only a finite number of which determine the value.

These two modifications give very simple and elegant syntax and semantics for Lucid, as presented in [1]. However, in practice, both for writing and proving programs, it is better to stick to the begin...end notation and not use latest.

Nesting introduces further restrictions on the use of assignment statements. If a variable is assigned to at one level, it can be referenced one level higher if the expression assigned is syntactically constant, i.e. built up from constants and first and as soon as expressions, by applying data operations. The variable cannot be referenced two or more levels higher. These restrictions are necessary to ensure that programs make sense, i.e. have unique "meanings".

6. Extensions

To make Lucid a useful programming language several additional features are needed. These include arrays, data structures and user-defined (possibly recursive) functions.

To some extent, adding new features to Lucid is non-trivial because as well as being natural to use as programming features, extensions must also fit smoothly with the existing formal semantics and be amenable to proof techniques within the (expanded) formal system.

On the other hand, the formal semantics is strictly denotational, and adding new features should not cause difficulties of a mathematical nature. The basis of Lucid should remain as described in this paper.

It is interesting to note that user-defined functions need not work pointwise. For example, we could define a function tot by

```
tot(x) = z
where
    first z = first x
    next z  = z+next x
```

If we called this function in, say, the program Prime, tot(I) would give us the sum of all the values of I so far, and tot(J) (in the inner loop of course) would give the sum of all the values of J so far in the current invokation of the inner loop. The efficient implementation of such functions might be difficult - here we would require that running totals be kept for future use, before the function is ever called.

7. Implementation

The implementation of Lucid is nontrivial for several reasons. Firstly, there is the necessity that the implementation be 'data-driven'; only such values should be computed as are needed in order to determine the value of output. More awkward however is the problem of 'time'. The formal semantics is given in terms of 'time-parameters', but, without complicated restrictions on the syntax of programs, the time parameters need not correspond with actual time. For example, we could write

X = if A > B then T else next X

so that, at any "time", X will be true provided A > B is going to be true some time "in the future". Moreover, the function as soon as gives us problems. Assume "X" is <false,false,false,true,... ,> then

"Y as soon as X" should give $\langle y_3, y_3, \ldots \rangle$

where y_3 is the value of Y at "time" 3 (the fourth element of "Y"). But "Y" may be <7,'undefined',2,5,...>, where each value of Y is the result of some inner loop say. Clearly, to compute Y as soon as X we must not try to compute Y until we have to, i.e. at "time" 3, or we will get stuck in the non-terminating second invokation of the inner loop. But to evaluate y_3 we may

have to use the values of other variables at "previous" times. At any given time we cannot decide what values of the variables will definitely be needed in the future, and it is unsafe to evaluate an expression unless its value is definitely needed because we may get stuck in a non-terminating computation.

The solution is to follow the formal semantics closely, treating the "time-parameters" simply as parameters, not as actual time. We start by demanding the value of output at some particular "time", which will require the values of other expressions at other "times". These subcomputations should proceed in parallel, demanding other values at other "times" until such values are found.

Two interpreters have been written, at Waterloo (Tom Cargill) and Warwick (David May) that follow this scheme. There is also a compiler (Chris Hoffmann) at Waterloo which treats time parameters as actual time (and which therefore can only correctly handle a subset of the language) which produces as fast code as, say, an Algol compiler.

8. References

[1] Ashcroft, E.A. & Wadge, W.W., "Lucid: a Formal System for Writing and Proving Programs". CS7501, Computer Science Dept., University of Waterloo.

[2] Ashcroft, E.A. & Wadge, W.W., "Program Proving Without Tears", Proc. Int. Symposium on Proving and Improving Programs, Arc et Senans (1975).