Lucid, a Nonprocedural Language with Iteration*

E.A. Ashcroft W.W. Wadge

Abstract

Lucid is a formal system in which programs can be written and proofs of programs carried out. The proofs are particularly easy to follow and straightforward to produce because the statements in a Lucid program are simply axioms from which the proof proceeds by (almost) conventional logical reasoning, with the help of a few axioms and rules of inference for the special Lucid functions. As a programming language, Lucid is unconventional because, among other things, the order of statements is irrelevant and assignment statements are equations. Nevertheless, Lucid programs need not look much different than iterative programs in a conventional structured programming language using assignment and conditional statements and loops.

1 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. It can be argued that one of the best proof methods, that of Hoare [4], is successful partly because the programming language statements are brought into the proof language. We want to take this process to its logical conclusion.

Our aim is to unify the two languages 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 [2]. 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.)

*Published as *Communications of the ACM* 20(7):519–526, July 1977. Preliminary version published as University of Warwick Computer Science Research Report CS-RR-O11, April 1976.

2 General Principles

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

$$\begin{aligned} root(n) &= s(0,1,n) \\ s(i,j,n) &= \texttt{if } j > n \texttt{ then } i \texttt{ else } s(i+1, j+2i+3, n) \end{aligned}$$

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

$$root(n)^2 \leq n \wedge n < \big(root(n)+1\big)^2.$$

The problem is that most programmers find the purely recursive approach too restrictive and thus are 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 (J.GT.N) GO TO 20
      J = J + 2*I +3
      I = I + 1
      GO TO 10
20    WRITE, I
      END
```

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:

$$\begin{aligned} N &= \texttt{first } input && (1) \\ \texttt{first } I &= 0 && (2) \\ \texttt{first } J &= 1 && (3) \\ \texttt{next } J &= J + (2 \times I) + 3 && (4) \\ \texttt{next } I &= I + 1 && (5) \\ output &= I \texttt{ as soon as } J > N && (6) \end{aligned}$$

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`.

But we can also consider the statements as true mathematical assertions about the *histories* of the variables. For example, statement (5) merely asserts that at each stage in the history of I

the value of I at the next stage is the current value of I plus one. More precisely, we define a history (or even better, a "world-line") to be an infinite sequence, i.e., a function with domain $\mathbb{N} = \{0, 1, \ldots\}$. The 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 $\langle x_0, x_1, x_x, \ldots\rangle$ 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 "$X \geq Y$" is also an infinite sequence and so may be neither "T" (each component true) nor "F" (each component false). Even equality is a pointwise operation; $X = Y$ is an infinite sequence of true values. The relation "*eq*" is "computable equality": if x_i and y_i are both undefined then the i-th value of X *eq* Y is also undefined, whereas the i-th value of $X = Y$ is true.

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 $(i+1)$-st 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).

We will say than an expression E is *constant* or denotes a constant if $E =$ `first` E. Note that `first` A and A `as soon as` B will be constant for any expressions A and B. Also all literals such as 3 and T denote constants.

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.

We can now say what we mean by a (simple) Lucid program. In general, a Lucid program is a set of equations specifying a set of variables. A variable V may be specified directly, by an equation of the form $V = E$, or inductively, by a pair of equations of the form

$$\begin{aligned} \texttt{first}\ V &= E_0 \\ \texttt{next}\ V &= E. \end{aligned}$$

In either case E and E_0 can be arbitrary expressions not containing "=" or "$\rightarrow$",[1] but the expression E_0 must be syntactically constant, i.e., built up from data constants, terms of the form `first` X or X `as soon as` P, and other variables that are equal to syntactically constant expressions, by applying data operations, relations and logical connectives. Every variable appearing in the program except "*input*" must be specified, and no variable may be specified twice.

This definition is very general, but nevertheless it can be shown by standard fixed-point methods that every program has a unique meaning in the following sense: no matter what value is given to the variable "*input*", there exists a minimal (least defined) solution to the program, considered as a set of equations. This solution consists of values for all the variables (including, possibly, undefined values). The value this solution gives to the variable "*output*" can therefore be considered as the output of the program. Note that the values of *input* and *output* are histories, that is, sequences of data values.

[1] The logical connective "$\rightarrow$" is implication, and, like "=", it is not computable because *undefined* $\rightarrow$ *undefined* is true.

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

$$\texttt{first}\,(X+Y) \quad = \quad \texttt{first}\,X + \texttt{first}\,Y$$

which allows us to reason about programs without referring explicitly to sequences. In fact, proofs can be made knowing very little of the formal semantics. We therefore proceed directly to a correctness proof of the sample program.

3 Sample Proof

Our goal is to derive the assertion $\mathit{output}^2 \le \texttt{first input} < (\mathit{output}+1)^2$ from the text of the Lucid program (considered as a set of assertions) together with the assumption

$$\mathit{integer}\,\texttt{first}\,\mathit{input} \wedge \texttt{first}\,\mathit{input} \ge 0.$$

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

$$\texttt{first}\,P,\ P \to \texttt{next}\,P \models P$$

(where for any assertion A and set Γ of assertions, $\Gamma \models A$ means that if everything in Γ is true, then A is true).

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

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

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

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

and so we have $\texttt{next}\,\big(J = (I+1)^2\big)$. Thus

$$\big(J = (I+1)^2\big) \to \texttt{next}\,\big(J = (I+1)^2\big)$$

is always true, since we were reasoning only about a single stage in the iteration. (This comment will be explained in the next section.) 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.)

Even though "=" is a pointwise relation, it is based on true equality, and when we prove $J = (I+1)^2$ it implies that J and $(I+1)^2$ are identical (have the same histories). Thus every occurrence of J in the program may be replaced by $(I+1)^2$. This produces the following equivalent "cleaned up" program:

$$\begin{aligned}
N &= \texttt{first}\,\mathit{input} \\
\texttt{first}\,I &= 0 \\
\texttt{next}\,I &= I + 1 \\
\mathit{output} &= I\ \texttt{as soon as}\ (I+1)^2 > N.
\end{aligned}$$

The value of *output* is the value I has when $(I+1)^2 > N$ is true for the first time; therefore, to prove that

$$output^2 \le N \wedge N < (output+1)^2$$

we prove that the analogous assertion is true of I when $(I+1)^2 > N$ is true for the first time, i.e., we prove

$$\big(I^2 \le N \wedge N < (I+1)^2\big) \texttt{ as soon as } (I+1)^2 > N.$$

To do this we use Lucid's `as soon as` induction rule:

$$\texttt{eventually } Q,\ \texttt{first } P,\ P \wedge \neg Q \rightarrow \texttt{next } P \models P \wedge Q \texttt{ as soon as } Q$$

with $P = I^2 \le N$ and $Q = (I+1)^2 > N$ (`eventually` Q is defined to be T `as soon as` Q). This rule states that if Q is eventually true, and if P stays true at least until that time, then P and Q are both true when Q is true for the first time. This rule is the Lucid analog of Hoare's `while` rule (see [4])

$$\frac{P \wedge \neg Q\ \{B\}\ P}{P\ \{\texttt{while } \neg Q \texttt{ do } B\}\ P \wedge Q}$$

except that it also deals with termination.

Simple calculations verify

$$\texttt{first } (I^2 \le N)$$

and

$$(I^2 \le N) \wedge \neg\big((I+1)^2 > N\big) \rightarrow (I^2 \le N)$$

so as to invoke the rule we need only establish

$$\texttt{eventually } (I+1)^2 > N.$$

This is the Lucid analog of proving termination for this program, and we use the basic Lucid "termination rule"

$$integer\ K, K > \texttt{next } K \models \texttt{eventually } K < 0$$

with $K = N - (I+1)^2$.

We can now apply the `as soon as` induction rule, giving

$$\big(I^2 \le N \wedge N < (I+1)^2\big) \texttt{ as soon as } (I+1)^2 > N$$

as desired. Finally we use axioms such as

$$(X+Y) \texttt{ as soon as } Q = (X \texttt{ as soon as } Q) + (Y \texttt{ as soon as } Q)$$

to "push" the `as soon as` past all the pointwise operations on the left; and then the rule

$$\texttt{eventually } Q \models \big((\texttt{first } X) \texttt{ as soon as } Q\big) = \texttt{first } X$$

and the fact that 1 and N are constants gives us

$$\big(I \texttt{ as soon as } (I+1)^2 > N\big)^2 \le N \wedge N < \Big(\big(I \texttt{ as soon as } (I+1)^2 > N\big) + 1\Big)^2.$$

Substituting "*output*" for "I `as soon as` $(I+1)^2 > N$" and "`first` *input*" for "N" we have

$$output^2 \le \texttt{first } input < (output+1)^2.$$

and this completes the proof. □

4 Proofs

The example has illustrated most of the axioms and rules for simple Lucid proofs. We will summarize these here. The formal system is given in detail in [2]. In some way it can be considered to be a tense logic, a sort of modal logic for expressing reasoning about time. The appropriateness of such logics for reasoning about programs has been recognized by Burstall [3].

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.

4.1 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

$$0 + 1 = 1 \quad \text{and}$$
$$\forall x \left[(x+2)^2 = x^2 + 4 \times x + 4\right].$$

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

4.2 Logical Properties

In the same way we can use most of the axioms and rules in inference of ordinary logic (e.g., from A and B infer $A \wedge B$). The few we cannot use fail either because we must allow an "undefined" truth value, or because we are talking about sequences. The law of the excluded middle fails because of the undefined truth value, and the deduction theorem fails because of sequences.

The first law, the law of the excluded middle, asserts that either A or $\neg A$ is true, and it fails because A may be undefined (the "result" of a computation that does not terminate).

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

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

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

The second rule, the deduction theorem, allows us to infer $A \to B$ ("A implies B") from a proof of B which has A as an assumption (this "discharges" the assumption). If 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 $I = 0 \to$ `next` $I = 0$ is not valid because $\to$ 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 of inference, like the induction rule, which depends on Lucid functions (Lucid *axioms* are admissible). This restricted sort of reasoning corresponds to considering one particular stage in an iteration.

4.3 Lucid Properties

4.3.1 Axioms

1. The most useful property is that `first` and `next` commute with data operations, relations and logical connectives so that, e.g.,

$$\begin{aligned} \texttt{first}\ (A + B) &= \texttt{first}\ A + \texttt{first}\ B \\ \texttt{next}\ (A + B) &= \texttt{next}\ A + \texttt{next}\ B \end{aligned}$$

2. The function as soon as commutes with those data operations, relations and logical connectives that are "strict", i.e., are undefined when all their arguments are undefined. (Note that "=" and "$\rightarrow$" are not strict.) For example,

$$X + Y \texttt{ as soon as } Q = (X \texttt{ as soon as } Q) + (Y \texttt{ as soon as } Q)$$

3.

$$\begin{aligned} \texttt{first first } X &= \texttt{first } X \\ \texttt{next first } X &= \texttt{first } X \end{aligned}$$

4.

$$\begin{aligned} \texttt{first } (X \texttt{ as soon as } P) &= X \texttt{ as soon as } P \\ \texttt{next } (X \texttt{ as soon as } P) &= X \texttt{ as soon as } P \end{aligned}$$

4.3.2 Rules of inference

$\texttt{first } P,\ P \rightarrow \texttt{next } P \models P$
$\texttt{eventually } Q,\ \texttt{first } P,\ P \wedge \neg Q \rightarrow \texttt{P} \models P \wedge Q \texttt{ as soon as } Q$
$\texttt{eventually } Q \models \texttt{first } X \texttt{ as soon as } Q = \texttt{first } X$
$\textit{integer } K,\ K > \texttt{next } K \models K < 0$

5 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. But programming in Lucid is in some ways quite different from conventional imperative programming, and the formal semantics outlined in Section 2 is not the best guide. We therefore present here a more informal, operational approach to the semantics of Lucid programs. This section is thus an explanation of, but *not* an addition to, the formal definitions of programs and their semantics given in Section 2.

In the following it will be realized that variables whose values are formally undefined give rise, in the operational approach, to nonterminating or abortive computations. Examples of such programs are $X = X$, $X = X + 1$ and $X = \texttt{next } X$.

The basic idea of the informal approach is to regard a Lucid program as built up from simple loops. The simplest loop consists of a single variable specified inductively in terms of itself and some constants; for example,

$$\begin{aligned} \texttt{first } V &= 1 \\ \texttt{next } V &= 2 \times V. \end{aligned}$$

We can interpret a loop like this as having the effect of first initializing the variable V, and then repeatedly reassigning to it, so that it takes on (in this case) the values $1, 2, 4, \ldots$. We call a variable which, like V, is inductively specified, a *loop* variable. Now in general we may have two or more loop variables specified in terms of each other, so that they form essentially a single loop. For example, if

$$\begin{aligned} \texttt{first } U &= 1 \\ \texttt{first } V &= 1 \\ \texttt{next } U &= V \\ \texttt{next } V &= U + V \end{aligned}$$

then U and V must be computed together, because the computation of the values of either U or V for the next iteration requires the current values of both U and V. It is important to realize the formal semantics implies that U and V are updated *simultaneously*. This point can be made clearer by introducing tupling into the language, so that the loop above can be rewritten as

$$\begin{aligned} \texttt{first}\ (U,V) &= (1,1) \\ \texttt{next}\ (U,V) &= (V, U+V). \end{aligned}$$

We can therefore interpret this loop as having the effect of first initializing the entire vector (U,V) of loop variables and then repeatedly reassigning to it, so that it takes on the values $(1,1)$, $(1,2)$, $(2,3)$, ...

A loop may have, besides the loop variables, other variables specified directly in terms of the loop variables. In the following loop

$$\begin{aligned} \texttt{first}\ V &= 1 \\ \texttt{next}\ V &= (V+W)/2 \\ W &= 100/V \end{aligned}$$

the variable W is of this type. These variables, which might be called "auxiliary variables", are essentially recomputed from the loop variables on each iteration, and so their values need not be carried over from one iteration to the next. In a sense the Lucid loop variables correspond to the index variables of an Algol `for`-loop, and the auxiliary variables correspond to variables local to a block within the Algol loop.

Values can be extracted from a loop using `as soon as`, as in the square root program, and the second argument of the `as soon as` can be thought of as the termination condition. A loop can use a constant specified elsewhere; we can therefore link loops by extracting a value from one and using it in another. This implies that the first loop has to be computed first, and the loops are *concatenated*. For example, if

$$\begin{aligned} \texttt{first}\ (I,J) &= (1,1) \\ \texttt{next}\ (I,J) &= \big(I+1, (I+1)\times J\big) \\ M &= J\ \texttt{as soon as}\ I\ \mathit{eq}\ 10 \\ \texttt{first}\ K &= M \\ \texttt{next}\ K &= K/2 \end{aligned}$$

then the result of the (I,J) loop is used to initialize the K loop, so that the letter cannot start until the constant M has been computed.

Besides concatenating loops, we can also nest them, as will be shown in the next section. We therefore have all the basic control structures of structured programming, except for the conditional statement. Lucid has no conditional statement, but it has the conditional expression (semantically, the ternary function `if–then–else` is just another function which, like addition, works pointwise on histories). Thus we can write the Euclidean algorithm for finding the *gcd* of two positive integer constants N and M as follows:

$$\begin{aligned} \texttt{first}\ (X,Y) &= (N,M) \\ \texttt{next}\ (X,Y) &= \texttt{if}\ X<Y\ \texttt{then}\ (X,Y-X)\ \texttt{else}\ (X-Y,Y) \\ \mathit{output} &= X\ \texttt{as soon as}\ X\ \mathit{eq}\ Y \end{aligned}$$

We see therefore that programming in Lucid is not completely different from programming in a conventional imperative language with assignment, conditionals and while loops. It is important to realize, however, that Lucid manages to treat assignment statements as equations, and to make loops implicit, only by imposing restrictions on the use of assignment. These restrictions all follow from the fact that a variable in a program can have only one specification, whether direct or inductive.

For a directly specified variable, the restriction is that the variable can be assigneed to at only one place in the program. For example, the two equations

$$\begin{aligned} X &= Y + 1 \\ X &= A \times B \end{aligned}$$

cannot both appear in the same program.

For an inductively specified variable, the restriction is that the variable can be assigned to only twice in the program, once for initialization and once for updating. For one thing, this means that a loop variable cannot be updated twice; thus the equations

$$\begin{aligned} \texttt{next}\ V &= 3 \times V \\ \texttt{next}\ V &= V + 1 \end{aligned}$$

cannot both appear in the same program. If an intermediate value is needed a separate variable must be used, e.g.,

$$\begin{aligned} V_1 &= 3 \times V \\ \texttt{next}\ V &= V_1 + 1. \end{aligned}$$

The restriction on inductive definition also means that every loop variable *must* be updated, whether the value is changed or not. The following statement cannot appear in a program:

$$\texttt{if}\ X < Y\ \texttt{then}\ \texttt{next}\ Y = Y - X$$

because it is not even an equation. Instead, we must write

$$\texttt{next}\ Y = \texttt{if}\ X < Y\ \texttt{then}\ Y - X\ \texttt{else}\ Y.$$

In a sense, Lucid allows only "controlled" or "manageable" use of assignment in much the same way as a conventional structured programming language allows only controlled or manageable use of transfer.

6 Nested Loops

A completely general approach to iteration must allow nesting of loops. For example, a program to test a positive integer N for primeness might, in a main loop, generate successive values 2, 3, 4, ... of potential divisors of N using a variable I, and in an inner loop generate for each value of I successive multiples of I using a variable J. It is sufficient to consider multiples of I starting with I^2; therefore for each value i of I the variable J should take on the values i^2, $(i+1)i$, $(i+2)i$, ...

Clearly, the history of the variable J is not simply a sequence of data items but rather a sequence of sequences. In the same way, the history of a variable defined at the third level of nesting is a sequence of sequences of sequences, that of a variable defined as the fourth level is a four-dimensional sequence.

We can make precise this more general concept of history by allowing functions of more than one integer time parameter. Thus in the program discussed above, the value of I would depend on only one parameter t_0, whereas that of J would depend on two parameters t_0 and t_1. To avoid messy type distinctions we define every generalized history to be a function of an infinite sequence $t_0 t_1 t_2 \cdots$ of distinct time parameters, i.e., a function from the set $\mathbb{N}^{\mathbb{N}}$ of infinite sequences of natural numbers into the set of data objects. We can assume, however, that any individual value of one of these histories is determined by only a finite number of these parameters. In particular, the value of a variable defined at nesting level n depends on only $t_0 t_1 \cdots t_{n-1}$ so that in a sense the remaining time parameters are "dummy variables" added for convenience. The *leftmost* time

parameters vary most rapidly. Thus t_0 is the number of iterations of the current loop, t_1 the number of iterations of the immediately enclosing loop, and so on.

Ordinary data operations are defined pointwise as before, and the special Lucid functions are defined as pointwise extensions to the 2nd, 3rd, etc., time parameters. Thus the value of $X + Y$ at "time" $\bar{t} = t_0 t_1 t_2 \cdots$ is the value of X at time $\bar{t}$ plus the value of Y at $\bar{t}$; the value of `next` X at time $\bar{t}$ is the value of X at time $(t_0 + 1)t_1 t_2 \cdots$; and the value of X `as soon as` P at time $\bar{t}$ is the value of X at time $st_1 t_2 \cdots$ where P is true at time $st_1 t_2 \cdots$ and false at time $rt_1 t_2 \cdots$ for all $r < s$ (if such an s exists). It is easily verified that the axioms and rules given in Section 4 remain valid under this new interpretation.

Note that the value of I corresponding to the $mn \cdots$ value of J is the $n \cdots$ value, since the number of iterations of the outer loop is n in this case. Thus to get the $(m + 1)n \cdots$ value of J we cannot just say `next` $J = J + I$, because this would take the $mn \cdots$ value of I. We must introduce a new Lucid function `latest` which gives us access to the extra time parameters such that the value of `latest` I at time $t_0 t_1 t_2 \cdots$ is the value of I at time $t_1 t_2 t_3 \cdots$. Then we say `next` $J = J +$ `latest` I, and everything works as desired. The effect of `latest` is to increase the number of time parameters upon which a history depends, so that, for example, if X is a function of t_0 and t_1 only, `latest` X depends on t_0, t_1 and t_2. The definition of programs given in Section 2 is modified to allow `latest` to be used in expressions, and to allow variables to be defined by equations of the form

$$\texttt{latest}\ V \quad = \quad A$$

provided the expression A is syntactically constant.

It is the function `latest` which permits nesting. For example, the program to test for primeness given below has a subloop which is 'invoked' for each value of I, and which on each invocation returns a value *IdivN* which is true iff some value of J is equal to N. This is done by referring in the subloop to the `latest` value of I (and N), and by defining the `latest` value of *IdivN* to be the result of the `as soon as`. Here is the complete program.

$$\begin{aligned}
N &= \texttt{first}\ \mathit{input} && (7)\\
\texttt{first}\ I &= 2 && (8)\\
\texttt{first}\ J &= \texttt{latest}\ I \times \texttt{latest}\ I && (9)\\
\texttt{next}\ J &= J + \texttt{latest}\ I && (10)\\
\texttt{latest}\ \mathit{IdivN} &= J\ \mathit{eq}\ \texttt{latest}\ N\ \texttt{as soon as}\ J \geq \texttt{latest}\ N && (11)\\
\texttt{next}\ I &= I + 1 && (12)\\
\mathit{output} &= \neg \mathit{IdivN}\ \texttt{as soon as}\ \mathit{IdivN} \vee I \times I \geq N. && (13)
\end{aligned}$$

Lines (9)–(11) constitute the nested loop. In general, using the `latest` value of a variable in a loop has the effect of making that variable "global" or "external" to a loop, and defining the `latest` value of a variable in a loop has the effect of "passing" it out of the loop.

The obvious problem with `latest` is that it clutters up programs and might be expected to clutter up proofs ever more. Fortunately, there is a convenient and suggestive "abbreviation" which allows us to avoid using `latest`. We introduce two special symbols, "`begin`" and "`end`", and allow assertions (statements) in a program to be placed between `begin`–`end` pairs nested to any depth. We interpret the enclosing of text by these `begin`–`end` brackets as having the effect of applying `latest` to every global variable in the text (a variable is global if it occurs outside the text). In other words, a `begin`–`end` pair can be removed if every occurrence of each global variable V in the enclosed text is replaced by "`latest` V".

Here is the above program written in `begin`–`end` form.

$N =$ `first` *input*
`first` $I = 2$
`begin`
  `first` $J = I \times I$
  `next` $J = J + I$
  $IdivN = J$ *eq* N `as soon as` $J \geq N$
`end`
`next` $I = I + 1$
output $= \neg IdivN$ `as soon as` $IdivN \vee I \times I \geq N$.

The `begin`–`end` notation permits a simple operational interpretation: inside a nested loop, the global variables are "frozen" (constant). Nesting using `begin`–`end` is therefore similar to nesting in an imperative language, but with two important restrictions: globals defined by equations outside a loop cannot be altered inside the loop, and globals defined inside the loop (like the variable *IdivN*) cannot be referred to two or more levels out. These restrictions are necessary to ensure that the result of removing the `begin`'s and `end`'s is a legal program, according to the rules given earlier.

Using our semantics we can derive rules with allow us to use `begin`–`end` and avoid `latest` in proofs as well as in programs. The first rule allows us to add to the statements in a loop anything derived from the loop statements plus the assumption that $V =$ `first` V for every variable V global to the loop. The second rule allows us to move in and out of a loop any assertions about global variables which do not have any occurrences of the Lucid functions. For example, in verifying the prime program we would first prove *integer* I and $I > 1$ outside the inner loop. Using the second rule, we move these assertions (plus certain assumptions about N) into the inner loop. Then inside the loop these assertions plus the assumptions $I =$ `first` I and $N =$ `first` N give us $IdivN = \exists k\ 2 < k \wedge k < N \wedge I \times k = N$. Since this contains no Lucid functions or local variables it may be brought back outside the loop and from it we eventually obtain *output* $= \exists m\ \exists k\ 2 \leq k \wedge k < N \wedge k \times m = N$. (For a more complete proof of this program, see [2].) This method of "nested proofs" permits reasoning about programs with nested loops to be reduced to reasoning about simple loops.

7 Implementation

Since Lucid has no prescribed *operational* semantics, a variety of methods of implementation are possible, depending on the data objects and operations used, and on the subset of the language to be implemented. One approach is to use analysis of the program in terms of loops (as described in Section 5) to translate the program into an imperative language, e.g., assembly code. Such a compiler has been written (in the language B) by Chris Hoffman at the University of Waterloo, and it runs under TSS on the Honeywell 6060. The compiled code runs in time comparable to that of equivalent Algol programs.

Another interesting possibility is to represent programs in terms of dataflow networks roughly of the type described in Kahn [5]. The advantage of this method is that independent computations can be carried out in parallel. Lucid is particularly well suited for this sort of implementation since there are no side effects, and the semantics imposes no 'sequencing' of computations other than that required by the data dependencies.

There are, however, two problems associated with either of these methods. The first is that not every legal program can be broken down into nested loops or translated into dataflow nets. Some of the worst examples arise when a variable is defined in terms of its own future. Thus the

equations

$$\begin{aligned} \texttt{first}\ N &= 7 \\ \texttt{next}\ N &= N - 1 \\ \mathit{output} &= \texttt{if}\ N \leq 1\ \texttt{then}\ 1\ \texttt{else}\ N \times \texttt{next}\ \mathit{output} \end{aligned}$$

form a legal program, the program has a minimal solution, and the solution gives as the value of *output* the sequence of values $\langle 5040, 720, 120, 24, 6, 2, 1, 1, 1, 1, \ldots \rangle$. It computes the factorial by "recursing" into the future. Bizarre programs like the above can be disallowed by simple syntactic restrictions.

The second more serious problem is that even with fairly conventional programs the compiled code or data net may perform more computation than necessary and the program may not terminate even though the formal semantics implies that *output* is defined. The function `as soon as` is responsible for most of these problems. For example, assume "X" is $\langle \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}, \ldots \rangle$, 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 $\langle 7, \mathit{undefined}, 2, 5, \ldots \rangle$, where each value of Y is, say, the result of some inner loop. 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 nonterminating second invocation 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 nonterminating computation.

These implementations are nevertheless partly correct, in that any defined value of *output* computed will be the value specified by the formal semantics. Furthermore, it is possible to formulate conditions under which these implementations are completely correct, although these conditions are not syntactic.

There is also a third, rather simple method, which is completely correct. The idea is to follow the formal semantics closely, making no attempt to interpret the time parameters as actual time. The implementation is instead demand driven, a demand being a request for some variable at some particular "time". The first demand is for the value of *output* at time 0, and this generates further requests for other variables at other times. This algorithm as it stands is very inefficient, but can be enormously improved by the simple strategy of storing some of the computed values for future use. Two interpreters have been written that follow this scheme, one at the University of Waterloo (Tom Cargill) and one at the University of Warwick (David May). An interesting feature of this method is that it can utilize extra store (for value of variables) as it becomes available, even in the middle of a computation.

8 Extensions

There are, besides iteration, several other features so far not discussed which a programmer would expect to find in a high-level programming language. These include arrays, structured data and user-defined, possibly recursive, functions. Naturally any such extensions must be compatible with the denotational approach; the addition of imperative features would make the rules of inference invalid. Function definitions offer no real difficulty, because, as was noted in Section 2, recursion equations are simply assertions. The addition of arrays is not quite so straightforward, but is possible if we allow the value of a variable to depend on space as well as time parameters (David May's interpreter deals with arrays in this way). Details will be given in a subsequent paper.

It is interesting to consider extensions that are suggested by Lucid itself, and which do not necessarily correspond to features in conventional imperative programming languages. For example, a useful addition is the binary function `whenever`:

$$\begin{aligned} X\ \texttt{whenever}\ P \;=\; & \texttt{if first}\ P \\ & \texttt{then first}\ X\ \texttt{followed by next}\ X\ \texttt{whenever next}\ P \\ & \texttt{else next}\ X\ \texttt{whenever next}\ P. \end{aligned}$$

The value of X `whenever` P is therefore the sequence consisting of those values of X for which P is true, so that, in the notation of Section 2, if P begins $\langle$*false*, *true*, *false*, *false*, *true*, ...$\rangle$, then X `whenever` P begins $\langle x_1, x_4, \ldots\rangle$. Note that X `as soon as` P is simply `first` (X `whenever` P).

Here is a simple program using `whenever`:

$$output \quad = \quad input \texttt{ whenever } (input\ ne\ \texttt{''} \vee \texttt{next } input\ ne\ \texttt{''}).$$

(The relational operator "*ne*" is computable nonequality, i.e., $x\ ne\ y$ means $\neg(x\ eq\ y)$.) If *input* is a stream of characters, *output* is the result of "compressing blanks" in *input*, i.e., the result of replacing consecutive blanks by single blanks. Note that this cannot be done with only the other Lucid functions, because we want to allow *input* and *output* to get arbitrarily out of step.

The following program (due to Gilles Kahn) uses `whenever` in the definition of a recursive nonpointwise function. (Defined *pointwise* functions are called "mappings"; see [1] for more details. The existence of non-pointwise functions is another interesting feature of Lucid.)

```
transformation sieve (I)
  first result = first I
  next result = sieve (I whenever (I mod first I) ne 0)
end
first N = 2
next N = N + 1

output = sieve (N)
```

The value of *output* is the infinite sequence of all prime numbers in increasing order.

References

[1] E. A. Ashcroft and W. W. Wadge. Lucid: Scope structures and defined functions. Technical Report CS–76–22, Department of Computer Science, University of Waterloo, November 1976. http://www.cs.uwaterloo.ca/research/tr/1976/CS-76-22.pdf.

[2] Edward A. Ashcroft and William W. Wadge. Lucid—A formal system for writing and proving programs. *SIAM J. Comput.*, 5(3):336–354, 1976. http://dx.doi.org/10.1137/0205029.

[3] Rod M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress*, pages 308–312, Amsterdam, 1974. North-Holland.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

[5] G. Kahn. A preliminary theory for parallel programs. Technical Report 6, IRIA, January 1973.