# THE SYNTAX AND SEMANTICS OF LUCID

E.A. Ashcroft*
&
W.W. Wadge**

Technical Report CS-84-24

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

August 1984

* Computer Science Laboratory, SRI International,
Menlo Park, California, U.S.A.

** Computer Science Department, University of Victoria,
B.C., Canada

# Table of Contents

# 1 Summary

The language Lucid has been developed, by the authors, over a period of ten years. In that time, the language has undergone many revisions, and, even, drastic changes in syntax. As a consequence, all of the published literature on Lucid (for example [1], [2], [3]) does not refer to the current version of the language. With the current version, the language has arrived, finally, at a stable, finished form. (Future changes should only be *extensions*, and existing programs should remain legal.) A book has been written about this final version [4], and it should appear in late 1984, or early 1985. This paper is concerned with the current language, and will, we hope, encourage the reader to study the book.

This paper will give a short description of the syntax and mathematical semantics of Lucid, followed by a discussion of operational ideas appropriate to the language. The first part demonstrates the formal simplicity of the language, and the second part demonstrates the power and expressiveness of the language and the subtlety of its possible operational interpretations. A short final section discusses possible extensions to the language.

# 2 Syntax

## 2.1 Introduction

Lucid is an expression language: Lucid programs are simply expressions (*terms*); there are no statements in the language. The language is definitional, and, apart from terms, the other main syntactic category is *definitions*.

Terms are built up using *constants*, *variables*, *operation symbols* and user-defined *function symbols*. (There are no procedures in the language.) Compound terms, called *where clauses*, are terms with subsidiary definitions. These clauses may contain declarations. (These declarations are not *type* declarations.)

Function symbols are defined in the same way as variables, as being the values of terms, but they also have *formal parameters*.

*Defined* variables are time-varying quantities. Some variables, namely the *declared* variables, are not time-varying. They are simply declared to be the current, or "frozen", values of expressions that are thought of as being evaluated outside the clause containing the declarations. As will be seen, this device gives the language the ability to specify subcomputations.

The variables and function symbols defined in a clause, together with the variables that are declared in the clause, are called the *locals* of the clause. The other variables and function symbols occurring in the clause are called the *globals* of the clause.

A *program* is simply a term in which every function symbol that is used is defined in some clause. The variables in the program that are not local to some clause are called the *input variables* of the program. (The value of the term is called the *output* of the program.)

### 2.2 Abstract Syntax

1. A *program* is a term;

2. A *term* is either
   a. a constant, or
   b. a variable, or
   c. an operation symbol together with *operands*, which are terms, or
   d. a function symbol together with *actual parameters*, which are terms, or
   e. a clause;

3. A *clause* is a *subject*, which is a term, together with a body;

4. A *body* is a set of declarations of distinct variables, and a set of definitions of distinct variables and function symbols;

5. A *declaration* is a variable (the *declared variable*), together with a *declarer* for that variable, which is a term;

6. A *definition* is a *right-hand-side*, which is a term and a *left-hand-side*, which is either a variable (for a variable definition), or a function symbol together with *formal parameters*, which are distinct variables (for a function symbol definition).

The term that is a program must be such that every function symbol appearing in it is defined in some clause.

Implicit in this syntax is the fact that the operands, actual parameters, and formal parameters are ordered. Also, function calls always have the same number of actual

parameters as there are formal parameters in the definition of the function symbol in question, and each operation symbol is always used with a particular number of operands, namely the "arity" of the operation.

### 2.3 Concrete Syntax

The following decisions concerning concrete syntax were made for the current Lucid implementation.

Nonmonadic operation symbols are written infix. Monadic operation symbols are written prefix (no parentheses necessary).

Function symbols are always written prefix, with parentheses around the actual parameter list, with the actual parameters separated by commas. The same convention is used in the left-hand-sides of definitions of function symbols, i.e., the function symbol precedes the list of formal parameters, and the formal parameters are enclosed in parentheses and are separated by commas.

Declarations have the words `is current` separating the declared variable and the declarer for that variable. Definitions are written as equations. Declarations and definitions are always terminated with a semicolon.

Clauses are written with the subject first and then the body. The body is enclosed by the words `where` and `end`, and the declarations are always written before the definitions.

### 2.4 Example

The following program (which computes the running root mean square of its input `a`) illustrates most of the features of the syntax of Lucid.

```
sqroot(avg(square(a)))
 where
   square(x) = x*x;
   avg(y) = mean
              where
                n = 2 fby n+1;
                mean = y fby mean + d;
                d = (next y - mean) / n;
              end;
   sqroot(z) = approx asa err < 0.0001
                where
                 Z is current z;
                 approx = Z/2 fby (approx + Z/approx)/2;
                 err = abs(square(approx)-Z);
                end;
 end
```

# 3 Semantics

## 3.1 Introduction

The primary semantics is denotational. Any implementation that produces outputs that are those specified by the denotational semantics is a correct implementation. The denotational semantics does not specify operational details, and there are several different ways of thinking operationally about Lucid programs, none of which might correspond to an actual implementation, but all of which might be helpful in understanding programs or designing programs. Some of these possible ways are discussed in the subsection about operational semantics.

## 3.2 Denotational Semantics

Lucid is actually a family of languages, not one particular language. Any one of these languages is differentiated from the others in the family by its choice of the data objects that programs can use, and the simple operations on these objects, and the constants, that are available as primitives. In other words, the language is characterized by an *algebra*. If the algebra is $A$, we will denote the corresponding version of Lucid by *Lucid(A)*.

We will consider an algebra to be a function from the symbols in its signature to their meanings, elements of the universe (carrier) of the algebra for constants, and operations over the universe of the algebra for operation symbols.

Because the denotational semantics of Lucid is given using fixpoint theory, the algebra on which a Lucid language is based has to be a *continuous* algebra. That is, the universe of the algebra must be a complete partial order (cpo), and all the operations in the algebra must be continuous (using the ordering of the cpo).

### 3.2.1 *Lu(A)*, The Algebra of Sequences from the Universe of *A*

Given a continuous algebra $A$, we can define a continuous algebra *Lu(A)* as follows. The signature of *Lu(A)* is the set of symbols in the signiture of $A$, together with various "Lucid operation symbols". These latter include the monadic operation symbol `next` and the dyadic operation symbol `fby` (which is read "followed by"). (Different members of the Lucid family may have different Lucid operators in addition to these, but they will all be definable in terms of `next` and `fby`.) The universe of *Lu(A)* is the set of all infinite sequences of elements of the universe of $A$. (We can consider an infinite sequence of elements of a set $S$ to be a function from the set of natural numbers into $S$.) The operation symbols, other than `next` and `fby`, are assigned, as meanings, operations on the universe of *Lu(A)* that are the pointwise extensions of the meanings given to the symbols by the algebra $A$. For example, in *Lu(A)* the addition of two sequences means simply the sequence of results produced by adding (according to $A$) corresponding elements of the two arguments.

The operation symbol `next` is assigned the operation *next* which, given a sequence, produces the same sequence but with the first element missing. (That is, the first element of *next(X)* is the second element of $X$, and so on.) The operation symbol `fby` is assigned the operation *fby* which, given two sequences, produces the sequence consisting of the second sequence but with the first element of the first sequence at the front. (That is, the first element of *fby(X,Y)* is the first element of $X$, and the second element of *fby(X,Y)* is the first element of $Y$, and so on.)

(It is easy to verify that *Lu(A)* is a continuous algebra if we take as the ordering on the universe of *Lu(A)* the pointwise extension of the ordering on the universe of $A$. That is, sequence $X$ is less than or equal to sequence $Y$ (by *Lu(A)*'s ordering) if and only if the $i$-th element of $X$ is less than or equal to the $i$-th element of $Y$ (by $A$'s ordering), for all $i$.)

The algebras *Lu(A)* will give meaning to the constants and operation symbols in *Lucid(A)* programs, but, in addition to constants and operation symbols, Lucid programs contain variables and function symbols. To give meanings to programs in a denotational manner, we must give meaning to the variables and function symbols. This is done by *environments*.

### 3.2.2 Environments

An *environment* is a function that maps variables into elements of *Lu(A)* and function symbols into operations on the universe of *Lu(A)*, of the appropriate arity. (For theoretical simplicity, the domain of an environment is the set of *all* variables and function symbols, but, in practice, only the variables and function symbols occurring in the program in question will be relevant.)

We now have all the mathematical apparatus necessary for the specification of the semantics of Lucid programs. To specify the meaning of a *Lucid(A)* program $P$, it is necessary to be given an environment that gives meanings to all of the input variables of $P$ (what it says about the other variables and the function symbols is irrelevant). Since a program is simply a term, to give meaning to programs it is sufficient to give a general definition of the meaning of terms, given an environment.

### 3.2.3 The Meaning of Terms

Assuming a continuous algebra $A$, the meaning $M(t,E)$ of a term $t$ in an environment $E$ is defined as follows.

1. If $t$ is a constant $k$, $M(t,E) = A(k)$.

2. If $t$ is a variable $x$, $M(t,E) = E(x)$.

3. If $t$ is an operation symbol $r$ together with operands $e_1$, $e_2$, .., $e_n$, $M(t,E)$ is $A(r)$ applied to $M(e_1,E)$, $M(e_2,E)$, .., $M(e_n,E)$.

4. If $t$ is a function symbol $f$, together with actual parameters $e_1$, $e_2$, .., $e_n$, $M(t,E)$ is $E(f)$ applied to $M(e_1,E)$, $M(e_2,E)$, .., $M(e_n,E)$.

5. If $t$ is a clause with subject $S$ and declarations $d$ and definitions $D$, $M(t,E)$ is defined a point at a time, as follows. The $i$-th element of $M(t,E)$ is the $i$-th element of $M(S,G)$, where $G$ is the least environment that agrees with $E_i$, except possibly for the locals defined in $D$, and satisfies all the definitions in $D$. The environment $E_i$ is like $E$, only the value of each variable, declared in $d$ to be the current value of $e$, is the constant sequence that is everywhere equal to the $i$-th value of $M(e,E)$.

This completes the definition of $M(t,E)$, apart from saying what it means for an environment to satisfy a definition. This is done in terms of $M$, so these two things are really defined simultaneously, mutually recursively. An environment $E$ satisfies a

definition of a variable $x$ with right-hand-side $e$ if the value of $x$ is $M(e,E)$. It satisfies a definition of a function symbol $f$ with formal parameters $g$ and right-hand-side $e$ if, for all environments $F$ that differ from $E$ only in the values given to the variables $g$, the value of the function symbol $f$, applied to the values given to $g$ by $F$, is $M(e,F)$.

The existence of least environments, which is needed for the cases when the term being evaluated is a clause, is guaranteed by an argument from fixpoint theory that depends crucially on the fact that the algebra in question is a continuous algebra.

### 3.2.4 Examples

1. First we will give definitions for several functions that, in practice, are really Lucid operations, in most Lucid interpreters.

   a.

```
first(s) = y where y = s fby y; end;
```

   This function simply returns a constant sequence, each element of which is the first element of the sequence that is its argument.

   b.

```
whenever(a,p) = if first p then first a fby x
                           else x fi
                where
                  x = whenever(next a, next p);
                end;
```

   This function gives the sequence of elements, taken from the sequence that is its first argument, for which the corresponding elements, in the sequence that is its second argument, are *true* (and no previous elements are other than *true* or *false*). If the sequence so specified is finite, the actual sequence is "padded out" with an infinite number of "undefined" elements, usually called *bottom*.

   c.

```
asa(a,p) = first whenever(a,p);
```

This function (which is pronounced "as soon as") simply gives the constant sequence, every element of which is the element of its first argument which is in the position at which the sequence that is its second argument is *true* for the first time (having previously been *false*). If there is no such position, the elements of the sequence are all *bottom*.

2.

```
s/n
  where
    s = j fby s + next j;
    n = 1 fby n + 1;
  end
```

This program has one input variable, j, and its meaning in an environment $E$ will depend on the value given to j by $E$. The value of the program will be the value of s/n in the environment $E'$ which differs from $E$ only in that it gives meanings to s and n that satisfy the definitions of these variables in the body of the where clause. If $E$(j), and hence $E'$(j), is the sequence <1, 2, 3, ...>, then $E'$(s) is the sequence <1, 3, 6, 10, ...>. (Note that $E'$(next j) is <2, 3, 4, ...>, so the claimed value of s satisfies the equation next s = s + next j, because + works pointwise in *Lu(A).*) Since $E'$(n) is <1, 2, 3, ...>, the value of the program is <1, 1.5, 2, 2.5, ...>. The program computes the running average of the values of j. The program could be used as the body of a function avg, as follows:

```
avg(j) = s/n
           where
             s = j fby s + next j;
             n = 1 fby n + 1;
           end;
```

This function, obviously, gives the running average of its argument. The function avg in Section 2.4 does the same thing, but in a better way. The function given here keeps a running sum of its argument, which could get to be a very large number, which could cause overflow. The function given in Section 2.4 avoids this problem.

3.

```
avg((a - M)*(a - M)) asa i eq 10
    where
     avg(j) = s/n
              where
               s = j fby s + next j;
               n = 1 fby n + 1;
              end;
     M = avg(a) asa i eq 10;
     i = 1 fby i + 1;
    end
```

This program contains the definition of the function avg given in 2. It also uses asa as a Lucid operation.

In this program, M is constantly the average value of the first ten values of the input variable a. The value of the program is constantly the average value of the squares of the divergences of the first ten values of a from the value of M. In other words, the value of the program is constantly the variance of the first ten values of a.

Notice how the average of the first ten values of a is subtracted from each of the values of a. This is expressed by a - M. The fact that M is a constant sequence is crucial here.

4. Example 3 above contained two occurrences of asa, one in the definition of M and one in the subject of the outermost where clause. Both of these give constant sequences. We have seen how the the fact that the sequence M is constant is crucial to the correctness of the program. What about the other asa, in the subject of the outermost where clause? Surely we do not need to endlessly repeat the same value; once would be enough.

To achieve this, all that is necessary is to replace the subject term by ((a - M)*(a - M)) asa I eq 10 fby eod. The constant eod stands for "end of data". Whenever the output of a program is eod, the execution of the program terminates. With this change, the program produces the variance of the first ten values of a and then terminates normally.

We could also change the program to give not just the variance of the first ten values of a, but the variance of all the values. That is, we can change the program to produce the running variance of a. All we have to do is successively replace the number 10 by the values 1, 2, 3, 4, etc., and recompute the variance each time. This can be done by using a **where** clause with a declaration.

```
avg((a - M)*(a - M)) asa i eq T
    where
     avg(j) = s/n
              where
               s = j fby s + next j;
               n = 1 fby n + 1;
              end;
     M = avg(a) asa i eq T;
     i = 1 fby i + 1;
    end
        where
          T is current t;
        end
            where
              t = 1 fby t + 1;
            end
```

With this program, to get its value at any time, say time $t$, we evaluate its subject term at time $t$. To evaluate the subject at time $t$, we evaluate *its* subject at time $t$, in an environment in which T is constantly the $t$-th value of t. *This* subject is just the program in Example 3, but with 10 replaced by T. Thus we successively compute the variances of one, two, three, etc. elements of a. Note that now it is important that the program in Example 3 produce a constant sequence, because different elements of these constant sequences are used as the values of the enclosing **where** clause.

5.

```
Power(p) = pow asa index eq N
              where
                N is current n;
                P is current p;
                pow = 1 fby pow * p;
              end;
```

This is an example of a function definition. It illustrates the conventional, and most common, use of the `is current` declaration. It also uses the Lucid constant `index`, which denotes the sequence <0, 1, 2, 3, ...>. In addition it illustrates the fact that functions can have global variables.

The function `Power` raises its argument to the power `n`. (The variable `n` is a global of the definition.) That is, the value of `Power(e)` at any time $t$ is the $t$-th value of `e` raised to the power $n$, where $n$ is the $t$-th value of `n`.

The function is quite conventional (but inefficient, since it calculates its result by repeated multiplication). It obeys the convention that anything, even zero, raised to the power zero is one.

6.

```
avg((a - M)*(a - M))
    where
      M is current avg(a);
      avg(j) = s/n
                where
                 s = j fby s + next j;
                 n = 1 fby n + 1;
                end;
    end
```

This program, like Example 4, gives the running variance of `a`. At any point, it gives the variance of the values of `a` up to that point. Clearly it is a much simpler program than Example 4.

Notice how the `where` clause works. At any point, the value of the program at that point is obtained by "freezing" at that point the value of the running average of `a`. (This is called `M`.) This is subtracted from *all* the values of `a` (not just from the value of `a` at the point in question), the results are squared, and a running average is kept of these squares. The value of the program is the value of this average at the point in question. This is the running variance of `a`. The program calculates essentially the same things as does Example 4, and uses them in the same ways, but makes much cleverer use of the `where` clause construct, and makes cleverer use of the `is current` declaration than does Example 5.

Neither this nor the earlier example are the best programs for computing the running variance. We haven't talked yet about the operational semantics of Lucid, but it should be clear already that this way of computing the variance gives an algorithm whose complexity at any point varies as the square of the number of elements of a being considered up to that point.

### 3.3 Operational Semantics

The semantics given Section 3.2 is mathematical. It gives very little intuitive idea of how programs would work, how they would behave. This is not an oversight; the mathematical semantics has priority. It is simple and reasonably elegant, and can be used to justify various program transformation rules and logical inference rules for program verification. However, the fact that the denotational semantics has priority when questions arise about the meanings of particular programs does not rule out the possibility of giving an operational semantics. In fact, such a semantics is very useful, as an aid to program design and program understanding.

Rather than attempt to give a complete, formal, operational description of the language, we will consider various example programs and give various ways of looking at the operational behavior of the programs.

There are three main features in Lucid that have operational significance:

1. The definition of variable values using `fby`

2. The different interpretations of the `where` clause construct

3. The interpretation of defined functions.

These will be considered separately below.

1. The crucial property of the `fby` operator is that it enables infinite sequences to be defined in a way that allows the computation of the sequence to be interpreted in iterative, operational terms. For example, the definition

   ```
   x = 1 fby x + 1;
   ```

   *could* be interpreted as as defining x to be the infinite sequence <1, 2, 3, 4, ...>, but it can also be interpreted in more operational, less declarative, terms, as saying that x starts off as 1, and then subsequently becomes 2, 3, 4, etc. For very many Lucid programs this is a better way to look at it. The language can then be interpreted in more operational terms, and it seems easier to design programs, partly because the reasoning necessary is not too

different, in this case, from that used in designing conventional imperative programs. (There are so many other features in the language that have unconventional operational interpretations that programmers do not have to feel that Lucid offers nothing new.) There *are* some cases where it is more appropriate to think of the values of variables as being infinite sequences. Usually, however, it is a mistake to think of Lucid as a language for manipulating, or talking about, infinite sequences.

2. The basic operational meaning of `where` clauses can be expressed as follows.

Consider the following general case of a simple `where` clause

```
E   where
      X is current e;
      y = t;
    end.
```

For each $i$, to get the $i$-th value of the clause, enough values of `y` have to be recomputed from the beginning to get the $i$-th value of `E`. When doing this, the value of `X` will be constantly the $i$-th value of `e`.

If there are no declarations, recomputing gives the same result as computing using the values of `y` that were previously computed when getting previous values of the clause. (This is because the least environments, which give the value of `y`, are the same in each case.) This means that `where` clauses can be interpreted as giving rise to subcomputations if they have declarations, and as giving rise to coroutines or parallel processes if they don't have declarations.

This can be seen in the examples we have considered already. The function `Power` of Example 5 in 3.2.4 has a `where` clause whose only free variables occur in declarations. This means that the function can be thought of as performing a subcomputation for each pair of values, for the argument of the function and for variable `n`. On the other hand, the function `avg` of Example 2 in 3.2.4 has a `where` clause without declarations. The function can be thought of as a coroutine or parallel process, activations of which are set up for each textual occurrence in a program of invocations of the function for

particular argument terms. These invocations run in parallel with the computations of the argument terms, whether or not the function has actually been invoked. For example, if a program contains the term

```
if y > 1000 then avg(x) + avg(x*x) else y + 1 fi
```

then there will be activations of the `avg` coroutine or parallel process for computing running averages of `x` and of `x*x`, right from the beginning of computation of values of `x`, even though these averages will only be ever needed if the value of `y` exceeds 1000.

It is important to realize the importance of the words "can be thought of as", in the preceding description. In fact, in all considerations of operational semantics for Lucid, the important thing is to find ways of thinking operationally about programs, *whether or not the Lucid implementation being used actually behaves in the way considered.* The operational ideas are aids to thinking about and designing programs. The actual implementation will probably work completely differently, because the ideas discussed here, and any which motivated programmers may come up with, will tend to be inadequate in some cases of Lucid programs.

3. The important operational idea for Lucid functions, apart from the way in which `where` clauses in the definition of the function affect the function's interpretation by the programmer, is that the arguments of functions are unending sequences of values. A function is thus a "continuously operating function". This fits best with the idea of a *filter*, a "black box" that is continually fed with input values, and that continually produces results (not necessarily at the same rate). This idea fits both the subcomputation and parallel process interpretations of functions.

### 3.3.1 Examples

The examples given in this section will just be examples that combine the two different ways of viewing computations of `where` clauses.

1.

```
s/(index + 1)
 where
  s = p fby s + next p;
  p = y asa index eq 10
       where
        X is current x;
        y = 1 fby y * X;
       end;
 end
```

This program simply computes the running averages of the tenth powers of the input variable x, that is, the running tenth moment of x. If the program were changed (or originally written) so that the inner `where` clause simply used x, rather than the current value of x, it would work completely differently. In fact, the inner `where` clause would simply give a constant sequence consisting of the product of the first ten values of x, and the whole program would give the running average of this sequence, which is, of course, the same sequence.

If we had a basic operation ** that performed exponentiation, the above program could be simply written as

```
avg(x ** 10)
```

provided we include a definition of `avg` such as the one in Section 2.4.

2. Example 6 in Section 3.2.4 is an example that has a `where` clause that has a declaration, but that also has an occurrence of a free variable, a, that is not within a declaration. There are two conceptually different ways of giving operational interpretations to such `where` clauses. One is to think of such clauses as 'basically' giving rise to subcomputations, but the free variables not occurring in declarations are thought of as being 'restarted' at the beginning of every subcomputation. Example 6 of Section 3.2.4 can be thought of this way, as can the clause defining `isprime` in the following program.

```
p
 where
  p = 2 fby nxtprime(p);
  nxtprime(q) = n asa isprime(n)
                where
                 Q is current q + 1;
                 n = Q + index;
                end;
  isprime(m) = farenough asa farenough or p | M
                where
                  M is current m;
                  farenough = p * p ge M;
                end;
 end
```

The program produces the sequence of all prime numbers. The function (filter) `isprime` can be thought of as testing successive values of its argument for primeness using a simple loop which runs through all the primes generated so far, starting with the first, and makes sure that none of them divides the current argument. (It is only necessary to check primes that are less than the square root of the current argument. After this point, the checking has gone "far enough". The current argument is prime if we have gone far enough through the list of primes without finding one that divides the current argument.)

3. The other way to view 'mixed' clauses is to consider them as parameterized coroutines or parallel processes, each set of current values of the declared variables yielding a particular coroutine or parallel process. The following program is an example of this.

```
avg(x ** N)
  where
   N is current n;
  end
```

This program has as its value, at time *t*, the n-at-time-*t*-th moment of the values of x up to time *t*. Think of the variable n as the parameter which yields running moment generators. For example, if n is constantly 2, the program is equivalent to

```
avg(x ** 2)
```

(which generates the running second moment of x), and, if n is constantly 3, it is equivalent to

```
avg(x ** 3)
```

(which computes the running third moments of x). If, instead, the value of n changes irreguarly with time between 2 and 3, the clause can be considered as sampling the appropriate outputs of two different simultaneously and continuously running processes computing the running second and third moments of x.

4. The same two interpretations can be applied to *functions* that *use* mixed clauses. In fact, as with clauses, it is possible to give two different interpretations to the same object. Consider, for example, the function defined as follows.

```
mom2(x,n) = avg((x - N)*(x - N))
              where
               N is current n;
              end;
```

The value of `mom2(a,k)`, at time $t$, is the second moment of the first $t+1$ values of `a` about the value of `k` at time $t$.

This function can be understood as an ordinary (but continuously operating) Algol-like function, except that its first argument, x, is restarted every time the function is called. It can also be understood as a parameterized set of parallel processes, with parameter n. Then `mom2(a,0)` is the running second moment of a, and `mom2(a,avg(a))` is the running variance. (This latter term can be viewed as a possibly infinite set of simultaneously running processes, one for each different value of the running average of a. The value of `mom2(a,avg(a))` at time $t$ is the value, at time $t$, of the process corresponding to the running average of a at time $t$.

All these different operational interpretations of Lucid programs are not intended to confuse the reader, but rather to illustrate the variety of different ways there are to consider Lucid programs. The real meanings of Lucid programs are given by the mathematical semantics, but these operational ideas, even if they seem a little impractical, often give a better idea of what programs mean, and give increased confidence in the correctness of programs.

# 4 Language Extensions

The language described so far is that implemented by the current Lucid interpreter, which is written in *C* and runs under Berkeley UNIX on a VAX. There are several improvements to the language that have been contemplated, and some of these will be considered here.

## 4.1 Arrays

Lucid is a family of languages. One way of adding a new "facility" to Lucid is simply to get a new instance of the family by using a richer and more complicated algebra on which to base the language. For example, this technique could be used to "add arrays to the language". This involves no change to the denotational semantics; all that is needed is that the elements of the algebra include arrays of some sort, and that there be operations that work on these arrays, in particular, indexing and updating operations. The updating operation, for example, would take as its arguments an array, the subscripts indicating the poition at which the array is to be changed, and the new element that is to be put into the array at that point. The result of the operation is the whole new array.

This approach will mean that some of the variables in programs will be "array variables", whose values, semantically speaking, will be infinite sequences of arrays. The approach will be impractical unless some way is found of having around only one instance, or possibly two, of the elements of such sequences. This can be done, in most cases, by making use of a storage management technique called "usage counts". (Details will not be given here.)

It is thus possible to have arrays in the language that can be altered one element at a time. This way of handling arrays does not really fit in very well with the general Lucid philosophy. Nevertheless, it *is* important that it is possible in Lucid to handle array updates at random points, if only because it can normally not be done in a dataflow language. Moreover, the method of making small changes to arrays can be adopted for use in a version of Lucid that can handle changes to list structures, using LISP's *rplaca* and *rplacd*, so this version of Lucid could be used in artificial intelligence applications.

Making single updates at random points is really not the best way of using arrays in Lucid, it is far better to use arrays in an APL-like style, in which whole new arrays are specified by applying operations to existing whole arrays. This approach will be considered in the following section.

### 4.1.1 Ferds

Lucid can be extended to give an 'extensional' way of using arrays, by making a more fundamental addition to the language, namely by actually changing its semantics (in particular, the definition of $Lu(A)$) to allow sequences that vary in space as the

'elements' of sequences that vary in time. This is the solution we describe here. The space-varying objects are called *ferds*. (The Oxford English Dictionary lists "ferd" as an obsolete word meaning "a warlike array".)

Although they are treated very similarly in the semantics, time and space are not quite symmetric. For one thing, time-varying objects are intended to be thought of in parts (in terms of individual values) whereas space-varying objects are thought of as a whole. Furthermore, although there is only one time parameter, there is no reason to have only one space parameter. In fact, multidimensional ferds are almost unavoidable. We want to allow streams of ferds and we want to be able to 'package up' any stream as a ferd. But when we package a stream of ferds, the result will be a ferd of more than one dimension.

This is the road we choose to follow. To avoid bothersome type distinctions, our ferds will all be notionally infinite dimensional.

The language Flucid ("ferd Lucid"), which we now introduce and describe informally, is based on an algebra-producing operation *Flu* in exactly the same way as Lucid is based on *Lu*. The elements of *Flu(A)* are streams of 'hyper-arrays' with individual values determined by one time parameter and infinitely many space parameters. In other words, they are functions from $N \times N^N$ to the universe of $A$ ($N$ being the set {0,1,2,...} of natural numbers).

Ordinary data operations are, of course, extended pointwise in space as well as in time. The Lucid operations, such as *next*, *fby* and *asa*, are extended pointwise in space.

The Flucid operations will generally deal specially with one particular space dimension. They will be subscripted with the number of the dimension being considered, but, if that number is zero, the subscript will often be omitted.

The basic Flucid operations act pointwise in time, but not in space. These are the space analogs of *first*, *next* and *fby*, called $initial_i$, $rest_i$ and $cby_i$ (continued by).

The definitions of $initial_i$, $rest_i$, and $cby_i$ are similar to those of *first*, *next* and *fby*, but thought of differently. Given a ferd $F$, $initial_i\ F$ is the first component of $F$ in the $i$-th dimension, the ferd obtained by taking a 'slice' in the $i$-th dimension at the first point; $rest_i\ F$ is the ferd resulting from dropping that component; and $x\ cby_i\ F$ is the result of sticking $x$ on to $F$ as the initial component in the $i$-th dimension. Thus, we have the properties

$$F = initial_i\ F\ cby\ rest_i\ F$$
$$initial_i\ (F\ cby_i\ G) = F$$

$$rest(F\ cby_i\ G) = G$$

Notice one important difference between the space operations and the corresponding time operations: $initial_i\ F$ is not the first component of $F$ stretched out in space in the same way that *first* $x$ is the first time component of $x$ stretched out in time. It is only the first slice of $F$ in the $i$-th dimension, and is not necessarily equal to $initial_i\ initial_i\ F$. The latter is the initial slice (in the $i$-th dimension) of the initial slice (in the $i$-th dimension) of $F$. In that sense, the three space operations correspond more closely to the list operations *hd* (*car*), *tl* (*cdr*) and *cons*.

Our ferds do not have any explicit rank, but an implicit rank can often be assigned. For example, the value of the individual components of $F$ may depend on the indices for the first three dimensions only. In such a case, $F$ can be thought of as 'really' just a three-dimensional object, an object of 'rank' three. The rank of $rest_i\ F$ is usually that of $F$, but the rank of $initial_i\ F$ is at least one less than that of $F$. If $x$ is rank $n$-1 and $F$ is rank $n$, then $x\ cby_i\ F$ is rank $n$ (whatever the value of $i$). Notice that we can define ferds of infinite rank; for example, with the definition

```
F = F cby1 2;
```

the object defined has a 'singularity' at the space point <0,0,0,...> (the value at this position is *bottom*, the least element of the data algebra $A$).

The next most important ferd operations are those for converting streams into ferds and vice versa. They do not work pointwise in either space or time. The operation $all_i$ takes a stream (of ferds) as its operand and returns as its result a ferd whose slices in the $i$-th dimension are the elements of the original stream; it converts time to space, so that if $j$ is the stream of natural numbers defined by

```
j = 0 fby j+1;
```

then $all_i\ j$ is the infinite vector of all natural numbers.

The value of $all_i\ x$ is independent of time; it is a constant. The companion of $all_i$ is $elt_i$ (element), and it converts space back into time. If $F$ is a ferd which is constant in time, $elt_i\ F$ is the stream of $i$-slices of $F$, i.e., the value of $elt_i\ F$ at time $t$ is the $t$-th slice of $F$ in the $i$-th dimension. If $F$ itself varies with time, $elt_i$ performs a 'gather', in that the value of $F$ at time $t$ is the $t$-th $i$-slice of the (ferd) value of $F$ at time $t$.

The two functions are 'not quite' inverses, in the following sense: we have

$$elt_i\ all_i\ x = x$$

for all values of $x$, but

$$all_i\ elt_i\ F = F$$

only if $F$ is constant. The operations $all_i$ and $elt_i$ are useful when we want to define a stream function in a manner more appropriate to arrays, or when an array is to be defined by generating its components in order.

From a conceptual point of view, ferds are easily incorporated in the informal iterative/dataflow operational model which we have been recommending as a programming guide. After all, one should have no difficulty in *imagining* infinite data objects flowing through a network. As for actual implementations, those which are based on demand-driven dataflow could easily be adapted.

Ferds are much more in the spirit of Lucid and dataflow than (say) the infinite lists used by Lisp-oriented functional languages. Moreover, this way of handling arrays appears to offer the best opportunities for exploiting parallelism in algorithms.

### 4.2 Types

The main reason why there are no types in Lucid as defined here is that most type systems have a semantics that is mathematically inelegant, compared to the current semantics of Lucid. This rather snobbish attitude is not really defensible, of course, and, in practical systems, Lucid programs will probably have types like most other languages. The lack of types is not a requirement or characteristic of Lucid.

### 4.3 Tuples

One very useful addition would be the addition of tuples to the language, so that a clause, and thus a function, could return a tuple of values, and not just a single value. The full generality of this construct can not be achieved using lists, because lists in Lucid are not lazy lists (we would want tupling to be lazy, i.e., we would want to be able to get a value out of a clause using a tuple even if the other values specified in the tuple were undefined), and because we would want to be able to have tuples on the left-hand sides of equations (which is not allowed for lists). There are some syntactic problems with tuples, of the type-checking variety, but, with some effort, these could be resolved more or less satisfactorily.

## References

[1] Ashcroft,E.A., and Wadge,W.W.
Lucid - A Formal System for Writing and Proving Programs.
*SIAM Journal on Computing* (3):336-354, September, 1976.

[2] Ashcroft,E.A. and Wadge,W.W.
Lucid, a Nonprocedural Language with Iteration.
*CACM* (7):519-526, July, 1977.

[3] Ashcroft,E.A. and Wadge,W.W.
*Structured Lucid.*
Technical Report CS-79-21, University of Waterloo, June, 1979.
Revised May 1980.

[4] Wadge,W.W. and Ashcroft.E.A.
*Lucid, the Dataflow Programming Language.*
Academic Press U.K., 1984.