Lucid: Scope Structures and Defined Functions

E.A. Ashcroft
Computer Science Department
University of Waterloo

and

W.W. Wadge
Computer Science Department
University of Warwick

CS-78-01

Lucid: Scope Structures and Defined Functions

E.A. Ashcroft
Computer Science Department
University of Waterloo

and

W.W. Wadge
Computer Science Department
University of Warwick

Abstract

In the paper we describe how Lucid can be extended to allow user-defined functions and scope conventions, i.e. conventions for limiting the range or scope of the validity of definitions. The new constructs added (they are called clauses) are similar in form to the blocks and procedure declarations of Algol-like languages, but are nevertheless strictly non-imperative, because a clause is actually a compound assertion, i.e. an assertion formed,as a program is, by combining a collection of assertions.

Each type of clause (there are four) has a straightforward mathematical semantics together with its own characteristic "manipulation rules" for general program massage. In addition, the informal operational view of (some) Lucid programs (described in a previous paper) can be extended to give an (incomplete) operational understanding of the effect of the clauses. In this framework a "compute" clause defines a block; a "mapping clause" defines a conventional (pointwise) function; a "produce" clause defines a block with persistent memory (an anonymous 'process' or 'actor'); and a "function" clause defines a general kind of coroutine.

INTRODUCTION

Lucid is a nonprocedural or denotative language; a Lucid program is an assertion or set of assertions defining its output, rather than a set of commands which some machine must obey in order to produce the output. Lucid is by no means the first such language; the distinction between imperative and denotative languages, and the advantages of the latter, have been well understood for at least a dozen years (see Landin [1]). Up until now, however, denotative languages have been almost universally regarded as elegant curiosities, the playthings of academics,unfit for "real" programming. The Lucid project is an attempt to demonstrate that this view is unwarranted, that denotative programming can be practical, and that in some respects it need not be so radically different from conventional "well structured" programming.

The first version of the Lucid language, (as described in [2], and henceforth referred to as "Basic Lucid") refuted some of these objection by showing that a "well-structured" form of iteration can be expressed very naturally in a denotative framework. But even more, we showed that it is indeed possible for denotative programs to have operational interpretations. The operational semantics for Basic Lucid (actually for a subset thereof) can be used informally as a guide to the programmer, but it can also be made precise, and used as the basis of an implementation, for example a compiler.

Of course, a programming language needs much more than a facility for iteration if it is to be practical, and in particular it needs facilities which allow programmers to restrict the scope of variables, and to define their own functions. In this paper we present an extension of Basic Lucid which has these features. The new constructs added (they are called clauses) are similar in form to the blocks and procedure declarations of Algol-like languages, but are nevertheless strictly non-imperative, because a clause is actually a compound assertion, i.e. an assertion formed, as a program is, by combining a collection of assertions.

Each type of clause (there are four) has a straightforward mathematical semantics together with its own characteristic "manipulation rules" for general program massage. In addition, the informal operational view of (some) Lucid programs (described in [2]) can be extended to give an (incomplete) operational understanding of the effect of the clauses. In this framework a "compute" clause defines a block; a "mapping clause" defines a conventional (pointwise) function; a "produce" clause defines a block with persistent memory (or an anonymous 'process' or 'actor'); and a "function" clause defines a sort of procedure with <u>own</u> variables (or a general kind of coroutine).

The treatment of these constructs naturally divides itself into two aspects, the formal and the informal, and this time we present both in the same paper. In the first part of the paper we introduce these new constructs, and informally describe their semantics, and indicate, through several examples, their use in programs. In the second, formal part of the paper we give the formal system and its semantics somewhat more precisely and present the manipulation rules for the clauses. These last are rules of inference, justified by the formal semantics, which allow us to transform programs or assertions which use clauses. We also show how these rules can be used to settle questions of calling conventions and variable binding for defined functions.

CLAUSES

Lucid has "constructs" for structuring programs analogous to the block, while-loops and procedure declarations of Algol-like imperative languages. These constructs are called clauses. Clauses are used in programs to define data, function or mapping variables, but, in the more general framework of the formal theory, a clause is a 'compound' assertion, i.e. an assertion built up by combining a collection of assertions.

Produce clauses

The simplest type of clause is the produce clause. A produce clause is used to limit the scope of certain variables so that the same variable can be used in different places with different meanings. A produce clause has the form

```
produce <data term > using <variable list>
                  <set of assertions>
end
```

Here is a simple example

```
          produce root using A,B,C
                 D      = B^2 - 4·A·C
                 output = (-B + √D)/(2·A)
          end
```

The < data term > at the head of the clause is the subject of the clause, the <variable list> is the global list, and the <set of assertions> is the body of the clause.

The variables occurring in the global list are the global variables of the clause. The special variable "output" (which must not appear on the global list) and any other variable not on the global list, but occurring free in an assertion in the body of the clause, is called a local variable. All other variables are unused by the clause. Semantically, a produce clause is an assertion about the subject and the globals* of the clause. Inside the clause the special local data variable "output" refers to the subject. The clause, considered as an assertion, is true iff there are values for the local variables which make all the assertions in the body true when "output" has the value of the subject term.

In the example given, "A", "B" and "C" are the global variables and "output" and "D" are the local variables.

---

* More precisely, it is an assertion about the values of the subject and globals.

If this clause occurs in a program, the definition of "D" in the clause is not valid outside it, although the definitions of "A", "B" and "C" are available inside the clause.

A produce clause is used in a program as a pseudo-equation defining its subject. When used this way, the body of the clause must (as in the example) be a subprogram, i.e. a set of equations and clauses defining output and other local variables in terms of each other and the globals (for more details see the next section). In particular, local variables can in turn be defined by other produces, i.e. produce clause can be nested, as in the following example

```
produce (X,Y) using $a_1,a_2,b_1,b_2,c_1,c_2$
   det = $a_1 \cdot b_2 - a_2 \cdot b_1$
   produce root1 using $b_1,b_2,c_1,c_2$,det
      g      = $c_1 \cdot b_2 - c_2 \cdot b_1$
      output = g/det
   end
   produce root2 using $a_1,a_2,c_1,c_2$,det
      g      = $a_1 \cdot c_2 - a_2 \cdot c_1$
      output = g/det
   end
   output = (root1,root2)
end.
```

This program solves the simultaneous equations

$$a_1x + b_1y = c_1$$

$$a_2x + b_2y = c_2 \ .$$

Although the definition of "det" is not valid outside the main produce clause, it *is* valid in the two inner clauses, because "det" is on the global list of each clause.

The meaning of a produce clause is independent of the choice of local variables. For example, we could have used "e" instead of "g" in one or both of the inner clauses.

The definition of the meaning of a produce clause given above works (makes sense) even when the body is not a subprogram. This is very important because it allows us to continue the Lucid practice of freely mixing program text with assertions in the course of verifying a program.

Function clauses

Function clauses are compound assertions about function variables, and are used in programs to define functions.

A function clause is of the form

```
 function <function term>  (<data variable list>)
                                  using <variable list>
       <set of assertions>
 end
```

Here is a typical function clause taken

program:

```
function Root(A,B,C)
   D      = B^2 - 4·A·C
   output = (-B + √D)/(2·A)
end
```

The function term (in this case, "Root") is called the subject of the clause, and the data variables enclosed in parentheses are called the formal parameters. The global variables are those in the global list (which in this case is empty, and the word "using" is dropped) and the locals are those occurring free in the body which are neither globals nor formal parameters. Formal parameters must all be distinct, and may not occur in the global list.

A function clause is an assertion about the subject and the globals. It asserts that, for all values of the formal parameters, there exist values of the local variables which make every assertion in the body of the clause true when "output" has the value of the subject applied to the formal parameters values.

When used in a program, a function clause is a definition of its subject, which must simply be a function variable, and the body of the clause must be a subprogram consisting of definitions of the locals.

A function(or mapping)variable can, like a data variable, appear on the global list of a clause, and thus a function(or mapping)definition appearing outside the clause can be valid inside. In particular, a function variable can appear on the global list of its own definition; this allows recursive function definitions. Here is the factorial function*:

---

* A definition of which, by law, must appear in any paper in which recursive functions are discussed.

```
function F(n) using F
   output = if n ≤ 1  then 1
                      else n·F(n-1)
end
```

Both these constructs are quite general and do not involve any of the functions in Lucid. They could be added to any assertional language to give facilities for restricting the scope of variables, and for defining functions.

Lucid has, in addition, two special versions of produce and function clauses, which allow the definition of subcomputations. They achieve this by implicitly applying latest to the global variables and latest^{-1} to the result (see [2]). These analogs of the produce and function clauses are the compute and mapping clauses respectively. In form they are identical to their analogs, except that the word "compute" replaces the word "produce", the word "mapping" replaces the word "function" and the subject of a mapping clause is a mapping expression. The terms "subject", "global", "local", "formal parameter" etc. are defined as for produce and function clauses. These clauses are used in programs to define their subjects, which in the case of a mapping clause must simply be a mapping variable. When used in a program, the body of compute or mapping clause must be a subprogram defining not the variable "output" but the special variable "result", which is synonymous with "latest output". Since the latest value of anything is quiescent (see [3]),"result" must be defined to be quiescent, e.g., by an expression of the form X asa P .*

Semantically, these clauses, like their analogs, are assertions about their subjects and their globals. A compute clause is true iff there exist values for the local variables which make all the assertions in the body true when each global has its latest value and "result" has the latest value of the subject. A mapping clause is true iff for all values for the formal parameters there exist values for the local variables which make every assertion in the body true when every global and formal parameter has its latest value and "result" has the latest value of the subject applied to the latest values of the formal parameters.

An Operational View

The mathematical semantics of clauses is simple and precise, even when stated formally, and is used to justify the rules of inference for reasoning about programs and to prove implementations correct. However, it is not the best guide for writing and understanding programs, because there is no notion of computations taking place or of anything "happening" at all. We therefore present an alternative,

* In this paper we will abbreviate as soon as by asa and followed by by fby.

operational view of the semantics of clauses in programs, which extends the operational view of Basic Lucid programs in terms of loops, described in [2]. This operational view is informal and is derived from the more basic mathematical semantics.

Of the four types of Lucid clauses, the simplest operationally (or at least the most conventional) is the compute clause. In a program, it is like an Algol block in that its subject is defined to be the result of a subcomputation carried out in a single step of the enclosing iteration, during which time the globals are considered to be 'frozen'. If the body of the clause has inductively defined variables, the subcomputation can be thought of as an iteration. For example, the following program statement

```
compute log10X using X
   I                      = 0 fby I+1
   P                      = I fby P·10
   first(A,B,logA,logB)   = (P,next P,I,next I) asa  next P > X
   compute C using A,B
      first R  = A·B/2
      next R   = (R + A·B/R)/2
      result   = R asa |A·B - R^2| < .00001
   end
   logC = (logA + logB)/2
   next(A,B,logA,logB) = if X < C then (A,C,logA,logC)
                                   else (C,B,logC,logB)
   result              = logC asa |A - B| < .00001
end
```

has a compute clause which defines each value of C to be (an approximation to) the corresponding value of $\sqrt{A \cdot B}$. This inner compute clause can be considered to be a nested loop which is run through completely on each step of the enclosing iteration. The compute clause replaces the Basic Lucid 'begin-end' construct.

The other clause with a fairly conventional operational interpretation is the mapping clause. In a program, a mapping clause defines a function guaranteed to be pointwise in its arguments and globals, i.e. a function whose value at a given point in time depends only on the value of its arguments and globals at that time. It does this because it freezes its parameters as well as its globals. Here, for example, is a definition of a mapping variable "trans"

```
mapping trans(L) using trans,dict
   result = if null(L) then NIL else
            if ¬ atom(L) then
                 cons(trans(car(L)),trans(cdr(L)))
               else L'
   D      = dict fby cdr(D)
   E      = car(D)
   entry  = E asa car(E) eq L ∨ null(E)
   L'     = if null(entry) then L
                                else cdr(entry)
end
```

which, when applied to the S-expression L returns the S-expression formed by replacing each atom by a corresponding S-expression, the correspondence being given by the pairlist dict. Because the definition of "trans" is recursive the function variable "trans" must appear on the global-list.

Terms involving mappings can be thought of as giving rise to Algol-like "mapping calls", where the parameter-passing mechanism is "call by need", as will be shown when we consider the manipulation rules for clauses.

The two types of clauses defined give the programmer roughly the facilities of Algol's blocks and procedures, with certain restrictions on side-effects. One of the reasons that these two clauses have a fairly conventional operational interpretation is that that in addition to restricting the scope of definition, they also freeze their globals and parameters so that they can be thought of as describing self-contained subcomputations. The produce and function clauses do not use this freezing effect and therefore their operational interpretation is completely different, because inner computations can interact with those of the enclosing iteration.

Operationally, the difference between a compute and a produce is that a produce clause must be considered either as an ongoing process which continuously produces values of its subject; or, alternatively, as a block of code which is repeatedly executed but with persistent internal memory in the form of inductively defined local variables.

For example, the following clause

```
produce Y using X
   N      = 1 fby N+1
   T      = first X fby T + next X
   output = T/N
end
```

defines the values of "Y" to be the "running averages" of the values of "X" up to that time, e.g. the third value of "Y" is the average of the first three values of "X". The local variable "T", for example, keeps a running total of the values of "X" . We must imagine either that the iterations of the body of the clause are running in parallel but in step with those of the enclosing iteration, or, alternatively, that the clause body is executed once on each step of the enclosing iteration, but the values of the local variables "I" and "T" are remembered between executions of the clause body.

Function clauses can be thought of as templates for processes, with each textual occurrence of a function call corresponding to the process which is the appropriate instance of the template. These processes must, like those defined by produce clauses, be thought of as operating in parallel, but synchronized with the enclosing iteration, and as updating internal variables even if, on some steps of the enclosing iteration, the output values are not required.

Alternatively, the function body can be thought of as a conventional Algol-like procedure body which is called and returns a result, provided in addition that (i) the inductively defined local variables are thought of as *own* variables whose values are remembered between one call and the next; (ii) different textual occurrences have separate copies of these variables; and (iii) the procedure is called on each step of the iteration containing the function call, *even when the value is not needed*, for "housekeeping purposes", namely to keep the *own* variables up to date.

For example, the following piece of program

```
N = 1 fby N+1
function Avg(X)
   I      = 1 fby I+1
   T      = first X fby T + next X
   output = T/I
end
Y = Avg(N)·Avg(N^2) asa N eq 5
```

defines Y to be 33, the average of the first five positive integers times the average of the first five squares. Whether we interpret this program in terms of processes or in terms of procedures with *own* variables the formal semantics requires first of all that the two occurrences of "Avg" make use of separate copies of "I" and "T", and secondly that the values of these variables be kept up to date even though no actual averages need be computed until the fifth step of the main iteration.

The operational view just described can be extended to cover recursive functions, but we must imagine that each recursive call sets up a new process, or generates new copies of the own variables. Some existing coroutine languages are capable of this, for example that of Kahn and McQueen [4].

The following are more subtle examples, using recursive functions. The examples are programs for computing with infinite formal power series with real coefficients. We consider a Lucid history P as representing the formal power series p whose coefficients are the successive values of P ; for example, if p is the power series $1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$ for e^x , then the value of P at time $t = 3$ is 1/6.

Now suppose that the functions "Prod" and "Div" are defined as follows:

```
function Prod(A,B)
   A0     = first A
   B0     = first B
   output = A0·B0 fby (A0·next B + B0·next A +
                  (0 fby Prod(next A,next B)))
end
function Div(A,B)
   q      = first(A/B)
   A1     = A - q·B
   output = q fby Div(next A1,B)
end
```

If A and B represent power series a and b respectively, then Prod(A,B) and Div(A,B) represent power series a·b and a/b respectively.

If we want to understand these programs operationally (and this is not necessarily a good idea) we must imagine that during execution more and more new activations of Prod or Div are produced. Of course, no Algol procedure, even with own variables, is capable of imitating such behaviour.

FORMAL TREATMENT OF CLAUSES

We now present a more precise treatment of Lucid considered as a formal system. We define the formal language of assertions and a special subclass, the class of legal programs, and we outline the formal semantics of assertions. We then consider questions of free and bound occurrences of variables, the substitution of terms for variables and so on, and present some important rules of inference which can be derived from the formal semantics.

Our emphasis will be on rules of inference, but not so much from the point of view of verifying programs as from the point of view of analyzing and manipulating them. In fact, the manipulation rules can be used, at least informally, to convey the meaning of Lucid's new constructs, and in particular to settle questions of 'calling conventions' and variable binding in regard to the evaluation of functions.

The Formal Language

The main extension to the formal language itself (i.e. the set of well formed terms and assertions) are (i) the addition of clauses; (ii) the addition of mapping and function variables and terms; (iii) the addition of tupling. By an assertion we will mean something that is either a term or a clause.

Clauses can be formalized using standard methods of abstract syntax, e.g. as in Landin [1].

Our mapping and function variables are used much as in first order logic. Technically speaking, they should be typed to indicate the number of arguments, but in practice we will omit any indication of type. Function and mapping terms are also typed and can appear anywhere a corresponding function or mapping variable could be used except in a global list. There are no higher order operations (operations which take functions or mappings as arguments) except for the polymorphic "latest" which can be applied to any term and yields a term of the same type. (Thus a mapping or function term must consist of a function or mapping variable with zero or more "latest"'s applied.)

The addition of tupling means that for each positive n we have in our language an n-ary tupling operation, and for each such n and each j less than n a projection operation p_j^n which selects the j-th component of a tuple of length n . Only data terms (i.e. not mapping or function terms) may be tupled, the result being a data term. We will use the bracket and comma notation for tupling, and not use the tupling operations explicitly.

Interpretations

As far as the semantics is concerned, the main changes are that we must allow tuples and also give meaning to mappings and functions.

If S is a standard structure (see [3]) we first close S out under tupling by taking the initial solution D of the domain equation

$$D = S + \bigsqcup_{n=1}^{\infty} D^n .$$

The operations on D (other than equality and if-then-else) are extended to D by defining their results to be $\bot$ whenever any of their arguments are tuples (there are therefore no coercions). Equality, if-then-else and the tuple constructors and projectors are interpreted in the obvious way (if-then-else yields $\bot$ unless its first argument is a Boolean and not a tuple).

The domain D plays the role previously played by S . Data variables and terms denote elements of Comp(D) , i.e. streams of objects which are possibly tuples of elements of S . N-ary mapping variables and terms denote elements of Comp($D^n \to D$) , i.e. streams of functions from D^n to D .

N-ary function variables and terms denote elements of $Comp(D)^n \to Comp(D)$, i.e. functions from streams to streams (which do not change with time).

Application of function terms to argument lists is interpreted in the usual way, but in applying a mapping term to its arguments we first take the values of the mapping and of the arguments at the point of time in question. For example, suppose that the 3-ary mapping term M denotes the element m of $Comp(D^3 \to D)$ and that data terms A, B and C denote elements a, b and c of $Comp(D)$. Then $M(A,B,C)$ denotes that element of $Comp(D)$ which at time $\bar{t}$ has the value $m_{\bar{t}}(a_{\bar{t}},b_{\bar{t}},c_{\bar{t}})$. In other words, mapping application is the pointwise extension of ordinary function application over D .

The function "latest" when applied to data or mapping terms is interpreted as described in [3], and when applied to function terms is interpreted as the identity operation (because functions do not change with time). Function terms (other than variables) are therefore in fact superflouous but are included for the sake of uniformity.

All other operations (they are data operations) are interpreted as in [3], so that, for example, the tupling operations over $Comp(D)$ are interpreted as the pointwise extensions of the tupling operations over D .

Following these outlines, it is straightforward to define $|E|_I$, the value associated with term E by the interpretation I. If E is a data term, $|E|_I$ is in Comp(D), if it is an n-ary function term it is in $\mathrm{Comp}(D)^n \rightarrow \mathrm{Comp}(D)$ and if it is an n-ary mapping term it is in $\mathrm{Comp}(D^n \rightarrow D)$.

Satisfaction

We can now say exactly what it means for an interpretation I to satisfy an assertion A (in symbols, what it means for $\models_I A$ to be true). Our definition merely makes precise the informal description of the semantics of clauses given in part 1.

Suppose that S is Comp(D), I is an S-interpretation and that A is an assertion.

If A is in fact a term, then $\models_I A$ iff $|A|_I$ is T_I. ($(T_I)_{\bar{t}}$ is true for all $\bar{t} \in N^N$.)

If A is a produce clause with subject E and globals $G_0, G_1, \ldots, G_{n-1}$, then $\models_I A$ iff there exists an S-interpretation I' such that

(i) $(G_m)_{I'} = (G_m)_I$ for any m less than n ;

(ii) $\mathrm{output}_{I'} = |E|_I$;

(iii) $\models_{I'} c$ for every assertion c in the body of A .

If A is a compute clause with subject E and globals $G_0, G_1, \ldots, G_{n-1}$, then $\models_I A$ iff there exists an S-interpretation I' such that

(i) $(G_m)_{I'} = |\text{latest } G_m|_I$ for any m less than n ;

(ii) $\text{output}_{I'} = |E|_I$;

(iii) $\models_I c$ for every assertion c in the body of A .

If A is a function clause whose subject is F , formal parameters are $V_0, V_1, \ldots, V_{k-1}$ and globals are $G_0, G_1, \ldots, G_{n-1}$ then $\models_I A$ iff for any sequence $v_0, v_1, \ldots, v_{k-1}$ of the universe of S there exists an S-interpretation I' such that

(i) $(V_i)_{I'} = v_i$ for any i less than k ;

(ii) $(G_m)_{I'} = (G_m)_I$ for any m less than n ;

(iii) $\text{output}_{I'} = |F|_I(v_0, v_1, \ldots, v_{k-1})$

(iv) $\models_I c$ for every c in A .

Finally, if A is a mapping clause with subject M , formal parameters $V_0, V_1, \ldots, V_{m-1}$ and globals $G_0, G_1, \ldots, G_{n-1}$, then $\models_I A$ iff for any sequence $V_0, V_1, \ldots, V_{m-1}$ of elements from the universe of S there exists an S-interpretation I' such that

(i) $(V_i)_{I'} = \text{latest}_S(v_i)$ for $0 \le i < m$

(ii) $(G_j)_{I'} = |\text{latest } G_j|_I$ for any j less than n ;

(iii) $(\text{output}_{I'})_{\bar{t}} = (|M|_I)_{\bar{t}}((v_0)_{\bar{t}}, (v_1)_{\bar{t}}, \ldots, (v_{m-1})_{\bar{t}})$ for all $\bar{t} \in N^N$;

(iv) $\models_I c$ for every assertion c in the body of A .

If Γ is a set of assertions and A is an assertion, as usual $\models_I \Gamma$ is true iff $\models_I B$ for all B in Γ and $\Gamma \models_I A$ is true iff $\models_I \Gamma$ implies $\models_I A$ for all Comp(D)-interpretations I .

Occurrences and Substitutions

Substitution plays a very important role in most formal systems, and Lucid is no exception. In order to define the result of substitutions, and the conditions under which they may be sensibly performed, it is necessary in addition to define the concepts of free and bound occurrences of variables.

The free occurrences of a variable V in a term T are as defined in [3]. The free occurrences of V in a clause c are either free occurrences in the subject, or, if V is in the global list of c, free occurrences in assertions in the body of c. A bound occurrence of V is an occurrence which is not free, and the free variables of a term or clause are those variables which have at least one free occurrence.

We can now define the result $A[\overline{V}/\overline{T}]$ of substituting a sequence of terms $\overline{T}$ for an equal length sequence of variables $\overline{V}$, of corresponding types, in a term or clause A. We will assume $\overline{V} = V_0, V_1, \ldots, V_n$ and $\overline{T} = T_0, T_1, \ldots, T_n$. The terms "local", "global" etc. are as defined earlier.

If A is a term the result is as defined in [3].

If A is a clause, $A[\overline{V}/\overline{T}]$ is the clause formed by replacing the subject E of A by $E[\overline{V}/\overline{T}]$ and in addition, if $\overline{V}'$ is the sequence of those variables in $\overline{V}$ occurring in the global list of A , by deleting $\overline{V}'$ from the global list of A , by adding all the free variables of $\overline{T}'$ to the global list of A , and by replacing every assertion B in the body of A by $B[\overline{V}'/\overline{T}']$.

The substitution just described will give unexpected results if some of the free variables of $\overline{T}$ are also locals of A , or if terms involving nonpointwise functions are substituted into the bodies of compute or mapping clauses. We must define what it means for variable V to be <u>free for term T in assertion A</u> .

If A is a term "freeness for" is as described in [3].

If A is a clause then V must be free for T in the subject of A , and in addition, if V is a global of A , none of the free variables of T can be locals or formal parameters of A and V must be free for T in every assertion in the body of A . Furthermore, if A is a compute or mapping clause, and V is a global of A , then T must be a <u>pointwise term</u>, i.e. must be built up from pointwise operations and mappings and not contain any function variables or Lucid operations.

For $\overline{V}$ a sequence of variables, $\overline{T}$ an equal length sequence of terms of the same types and $\overline{A}$ a <u>set</u> of assertions, we say that $\overline{V}$ is free for $\overline{T}$ in $\overline{A}$ provided each V_i is free for the corresponding T_i in every assertion in $\overline{A}$.

To illustrate these ideas, consider the example mapping clause given earlier. The free variables of the clause are "trans" and "dict". The variable "dict" is free for cons(I, cdr(J)) , but not for "car(D)" or "cons(Q,L)" or "cons(next I, cdr(J))" , in the clause. The result of substituting "cons(I, cdr(J))" for "dict" is the following:

```
mapping trans(L) using trans,I,J
    result = if null(L) then NIL else
                 if ¬ atom(L) then
                    cons(trans(car(L)),trans(cdr(L)))
                    else L'
    D      = cons(I,cdr(J)) fby cdr(D)
    E      = car(D)
    entry  = E asa car(E) eq L ∨ null(E)
    L'     = if null(entry) then L else cdr(entry)
end .
```

Simple Programs

A Lucid program is an assertion about the data variable "output" which can be understood operationally as describing an algorithm which, given values for the program's designated input variables, allows one to compute a value for "output" which makes the program a true assertion. To avoid indeterminancy, we consider only programs that are strictly definitional, i.e. those which are built up from definitions of variables. In this case, the algorithm described by the program is determinate, and leads to a single value of the variable "output" which constitutes the least defined solution.

A definition program is simple if every definition in it is direct, i.e. an equation whose left hand side is a data variable, or a clause whose subject is a variable.

To define simple programs more precisely, we need a few subsidiary definitions.

The subject of an equation is its left hand side; its globals are all the variables occurring freely on the right hand side.

A simple subprogram is a set of simple statements, no two having the same subject. The subjects of the subprogram are those variables which are subjects of statements in the program, and the global variables are those, other than the subject variables, which are globals of some statement in the program. One of the subjects must be the variable "output".

A simple statement is

(a) an equation whose subject is a data variable and whose right hand side is a data term

or (b) a clause whose subject is a variable and whose body is a simple subprogram P such that

(i) every global variable of P is a formal parameter or a global of the clause and (ii) no subject variable of P is a formal parameter or a global of the clause.

A simple program is a simple statement whose subject is the data variable "output".

The fact that every simple program has a unique least defined solution follows from the following lemma; given any simple subprogram P and values for the global variables, there are unique least-defined values for the subjects of P which make all the assertions in P true. This lemma is proved by induction on the structure of P , using standard fixpoint techniques.

Thus we see that every simple program is "meaningful" in that it has a unique least defined solution. Some of these programs, however, are very ill-behaved in that the time-parameters get so intermingled that we cannot separate out levels of nesting of subcomputations. This happens either because some compute or mapping clauses fail to unfreeze their output, or because **latest** and $\textbf{latest}^{-1}$ are used indiscriminately.

We can guarantee that a simple program is well-behaved provided (i) in every compute or mapping clause output is defined by an equation whose right hand side is of the form $\textbf{latest}^{-1}$ E ; and (ii) there are no other occurrences of $\textbf{latest}^{-1}$ and none of **latest** at all.

We can ensure that a program is well-behaved without using $\textbf{latest}^{-1}$ by requiring that "output" in any compute or mapping clause be defined by a statement whose subject is "**latest** output" (this will involve checking, as described below, that the right hand side is quiescent, because the **latest** value of anything must be quiescent). We can also eliminate **latest** from programs by introducing the special variable "result" which is defined to be synonymous with "**latest** output" and using it instead of "output" in mapping and compute clauses. Programs which use only "result" in compute and mapping clauses are called *orthodox*, and they are guaranteed to be well-behaved.

General Programs

Simple programs are very restrictive because each definition directly defines exactly one variable. Thus, they do not allow indirect or multiple definitions.

An indirect definition of a variable consists of possibly several definitions of various aspects of the variable. The simplest example is the definition of a variable X , say, by defining **first** X and **next** X separately.

A multiple definition is the use of tuples to define two or more variables simultaneously, for example,

(A,B) = *if* X > Y *then* (C,D) *else* (E,F) .

Both sorts of definition can be combined, as in

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

Programs with such definitions can be "massaged" into simple programs by using the projection operations and the function fby, for example

```
        first X = E0
        next X  = E1
becomes X = E0 fby E1
and     (X,Y,Z) = W
becomes X = p^3_0(W)
        Y = p^3_1(W)
        Z = p^3_2(W) .
```

The problem is that the simple program P' which results from massaging a program P is guaranteed to have a unique minimal solution (since the projection operations and fby are continuous) which may not be a solution of P (because P might have no solution at all). The reason is that the more general sorts of definitions can only be valid under certain conditions, whereas no conditions are imposed on the existence of a solution for simple programs. In particular, first $X = E_1$ implies that E_1 is quiescent (see [3]), and $(X_1, \ldots, X_n) = E_2$ implies that the value of E_2 is an n-tuple. We cannot simply take the minimal solution of P' to be the meaning of P regardless of whether or not it is a solution of P because we wish to prove properties of P from assertions in P . If P has no solution, it is therefore inconsistent and we would be able to prove anything from it. For the same reason, we must deduce the conditions for the existence of a solution for P from P' rather than from P itself.* This can be done by a sort of type-checking on P' , pushing our original conditions back through P' , generating new conditions in the process. For example, if A must be quiescent, and A = B + C then both B and C must be quiescent. For details of the general process, see Wadge and Ashcroft [7]. Using this technique, we can allow general programs to have tuples of tuples etc., and even self-referential tuples, as in

$$X = (a, X) .$$

Surprisingly, it can nevertheless be shown that the projection operations in P' can be pushed through other operations until both projection operations and tuples are completely removed, unless the output of the program is itself defined to be a tuple.

* To paraphrase a famous logician, it would otherwise be like having the defendant at a perjury trial testify on his own behalf.

Rules of Inference

When reasoning about programs we do not use the formal semantics directly but rather use rules of inference whose validity follows easily from the semantics.

Following the natural deduction style of [3], there are two basic rules for each type of clause, one for introduction and one for elimination.

Produce Clauses

(PI) For $\bar{L}$ a sequence of distinct variables including the data variable "output", $\bar{E}$ an equal length sequence of terms of the corresponding types, with e_0 corresponding to "output", Δ a set of assertions such that $\bar{L}$ is free for $\bar{E}$ in Δ, and $\bar{G}$ a sequence of variables disjoint from $\bar{L}$

$$\Delta[\bar{L}/\bar{E}] \models P$$

where P is

```
produce e_0 using G̅
    Δ
end
```

(PE) If Γ and Δ sets of assertions and A an assertion, and "output" is free for data term e_0 in Δ and $\bar{G}$ is the set of all variables occurring free in Δ and also in either Γ or A

```
    if  $\Gamma, \Delta[\text{output}/e_0] \models A$
      then  $\Gamma, P \models A$
  where  P  is
      produce $e_0$ using $\overline{G}$
         $\Delta$
      end
```

Function Clauses

(FI) If $\overline{X}$ is a sequence of data variables and $\overline{G}$ is a disjoint sequence of variables, and Γ is a set of assertions in which no variable in $\overline{X}$ occurs free, and f is a function term

```
      if $\Gamma \models P$ then $\Gamma \models F$
  where  P  is
      produce $f(\overline{X})$ using $\overline{G},\overline{X}$
         $\Delta$
      end
  and  F  is
      function $f(\overline{X})$ using $\overline{G}$
         $\Delta$
      end
```

(FE) Let $\overline{X}$ and $\overline{G}$ be disjoint sequences of distinct variables and F be

```
      function $f(\overline{X})$ using $\overline{G}$
         $\Delta$
      end
```

and P be

```
    produce f(X̄) using Ḡ,X̄
        Δ
    end
```

For $\overline{E}$ a sequence of terms of the same length and type as $\overline{X}$ such that $\overline{X}$ is free for $\overline{E}$ in P

$$F \models P[\overline{X}/\overline{E}] \quad .$$

Compute Clauses

(CI) For $\overline{L}$, $\overline{E}$, e_0, Δ and $\overline{G}$ as in (PI),

$$\Delta[L/E,\ \overline{G}/\text{latest}\ \overline{G}] \models C$$

where C is

```
    compute e0 using Ḡ
        Δ
    end
```

(CE) For Γ, Δ, A, e_0, $\overline{G}$ as in (PE)

if Γ, $\Delta[\text{output}/e_0,\ \overline{G}/\text{latest}\ \overline{G}] \models A$

then Γ, $C \models A$

where C is

```
    compute e0 using Ḡ
        Δ
    end
```

Mapping Clauses

(MI) and (ME) are identical to (FI) and (FE) respectively, with the word "mapping" replacing the word "function" and the word "compute" replacing the word "produce".

Program Transformation

The rules just given are elegant and concise but are not the most practical. For one thing they force the verifier to explicitly introduce and reason about the function *latest*. For another, proving even simple properties usually involves taking the program apart using the elimination rules and putting it back together with the introduction rules. Fortunately, there is a collection of derived rules which allow a "nested proof" technique as described in [3]. These rules are transformation rules, rules (like the (FE) rule) of the form $A \models A'$ where A' is the result of making some "small" change to A, for example adding a variable to a global list or moving an assertion out of some subclause. One consequence is that a correctness proof of a program P can proceed by linear reasoning [5], i.e. it consists of a sequence $A_0, A_1, \ldots, A_n$ of assertions where $A_0 = P$, each A_{i+1} follows from A_i by using some rules and A_n is the desired statement of correctness.

The first two rules allow us to carry on normal reasoning within clause boundaries. It says that we can add to any subclause any assertion that follows logically from the assertions in the subclause. Conversely, the second rule says that we can throw away any assertion from any subclause.

There is also a set of "movement rules" which allow us to move across clause boundaries assertions whose free variables are all globals of the clause. Any such assertion can be moved in or out of a produce or function clause, and any such assertion which is pointwise* can be moved in or out of a compute or mapping clause. Furthermore in the case of a produce clause, assertions which refer to "output" can be moved out of the clause provided "output" is replaced by the subject of the clause, and conversely, assertions which refer to the subject can be moved in provided the subject is replaced by "output". Similarly, the same is true of compute clauses, if we consider "result" rather than "output".

These are the most important rules because they allow both small and large changes to an assertion in a single step - the assertions being moved can themselves be clauses. In order to "prepare the stage" for the application of these rules we also need rules for adding global variables to a clause and for renaming its local variables. These rules can be easily derived from the rules of inference.

While these rules are natural and easy to use, we know of no small well-structured symmetric subset of them

---

* An assertion is pointwise if it is a pointwise term or a compute or mapping clause whose subject is a pointwise term or a function clause with no global variables.

which are in some sense complete.

"Computational Behavior" of Functions and Mappings

The formal definitions of clauses and the manipulation rules above are sufficient to answer questions of an operational nature about functions and mappings.

We will first illustrate how the rules can be used to perform symbolic execution of a function call. Consider

```
produce M
   function Min(X)
      first output = first X
      next output  = if output < next X then output
                                         else next X
   end
   Z      = 3 fby 1 fby 2
   output = Min(Z^2)
end
```

We first use the (FE) rule, with the formal parameter X being replaced by Z^2, yielding

```
produce M
   produce Min(Z^2) using Z
      first output = Z^2
      next output  = if output < next Z^2
                        then output else next Z^2
   end
```

```
      Z      = 3 fby 1 fby 2
      output = Min(Z^2)
   end
```

Then we use our movement rule to move the definition of Z inside the produce clause yielding

```
   produce M
      produce Min(Z^2) using Z
         first output = first Z^2
         next output  = if output < next Z^2
                              then output else next Z^2
         Z = 3 fby 1 fby 2
      end
      output = Min(Z^2)
   end
```

Then inside the produce we substitute "3 fby 1 fby 2" for every occurrence of "Z" , perform some simple calculations and, after discarding unnecessary statements,we have

```
   produce M
      produce Min(Z^2) using Z
         output = 9 fby 1 fby 1
      end
      output = Min(Z^2)
   end
```

The assertion "output = 9 fby 1 fby 1" can be moved out of the inner produce, yielding "Min(Z^2) = 9 fby 1 fby 1". Then substitution of equals for equals yields "output = 9 fby 1 fby 1" in the body of the outer produce and this can be brought out,giving "M = 9 fby 1 fby 1".

Any mechanism to implement functions and mappings must produce effects that are consistent with all properties that can be proved using the manipulation rules. In particular, one such parameter-passing mechanism is the call by name rule as considered in Vuillemin [6]. To see that call by value does not work, consider

```
produce V
   mapping f(X,Y) using f
      result = if X eq 0 then 0 else f(X-1,f(X,Y))
   end
   output = f(1,0)
end
```

We can duplicate the mapping clause, and then replace one of the copies by the corresponding compute clause (setting up the actual/formal parameter correspondence) giving

```
produce V
   mapping f(X,Y) using f
      result = if X eq 0 then 0 else f(X-1,f(X,Y))
   end
```

```
      compute f(1,0) using f
         result = if 1 eq 0 then 0 else f(1-1,f(1,0))
      end
      output = f(1,0)
   end
```

Inside the compute clause we obtain

```
                  result = f(0,f(1,0))
```

which can be moved out, and the compute clause discarded, giving

```
   produce V
      mapping f(X,Y) using f
         result = if X eq 0 then 0 else f(X-1,f(X,Y))
      end
      f(1,0) = f(0,f(1,0))
      output = f(1,0)
   end .
```

We repeat the process with formal parameter X being replaced by 0 and Y being replaced by f(1,0) . Inside the resulting compute clause we get

```
      result = if 0 eq 0 then 0 else f(0-1,f(0,f(1,0))) .
```

This simplifies to

```
                  result = 0
```

and when we move this out and discard the compute clause we get

```
produce V
   f(0,f(1,0)) = 0
   f(1,0)      = f(0,f(1,0))
   output      = f(1,0)
end .
```

From this we clearly get

output = 0

which we can move out giving

V = 0 .

In a call by value implementation of this function, the program would diverge, which is inconsistent with the fact that "V = 0".

A more efficient mechanism than call by name is the "delay rule" or "call by need" of Vuillemin [6], and Lucid may be the first programming language that can actually use it.

It is also worth noting that implementing <u>non-recursive</u> functions and mappings is no more difficult than implementing produce and compute clauses, since the latter can be used as 'macro expansions' of "calls" of functions and mappings.

Finally, the transformation rules can be used to settle the question of dynamic versus static binding of global variables of functions and mappings. Consider

```
compute U
   mapping f(X) using Y
      result = X + Y
   end
   Y = 1
   compute result using f
      Y      = 2
      result = f(3)
   end
end .
```

The question is whether the value of U is 5 or 4, which depends on whether the inner or outer definition of Y is used in the evaluation of f(3) . In a language like LISP which has dynamic binding, the inner value would be used, in Algol the outer. It is easy to see that the manipulation rules imply that Lucid uses static binding. We cannot move the mapping inside the inner compute clause until we have renamed the inner local variable Y . The variable Y can then be added to the global list of the inner compute, and the mapping can be moved giving

```
compute V
   Y = 1
   compute result using Y
```

```
        mapping f(X) using Y
           result = X + Y
        end
        Z     = 2
        result = f(3)
     end
  end
```

The definition of Y can be brought down into the mapping, and it is then straightforward to finally obtain "V = 4".

REFERENCES

1. Landin, P.J. The next 700 programming languages. Comm. ACM. 9, 3(March 1966) 157-166.

2. Ashcroft, E.A., and Wadge, W.W. Lucid, a nonprocedural language with iteration. Comm. ACM 20, 7(July 1977), 519-526.

3. Ashcroft, E.A., and Wadge, W.W. Lucid : a formal system for writing and proving programs. SIAM J. Comptg. 5, 3(Sept. 1976) 336-354.

4. Kahn, G., and MacQueen, D. Coroutines and networks of parallel processes. Res. Rep. No. 202, IRIA, France, Nov. 1976.

5. Craig, W. Linear reasoning, a new form of the Herbrand-Gentzen theorem. J. Symbolic Logic, 22 (1957) 250-268.

6. Vuillemin, J. Correct and optimal implementations of recursion in a simple programming language. Res. Rep. No. 202, IRIA, France, July 1973.

7, Wadge, W.W., Ashcroft, E.A. The type checking of recursive list programs. In preparation.