LUCID: SCOPE STRUCTURES AND DEFINED FUNCTIONS

by

E.A. Ashcroft
Dept. of Comp. Science
University of Waterloo
Waterloo, Ont., Canada

W.W. Wadge
Dept. of Comp. Science
University of Warwick
Warwick, Warwickshire

Research Report CS-76-22

November 1976

LUCID: SCOPE STRUCTURES AND DEFINED FUNCTIONS

by

E.A. Ashcroft
Computer Science Department
University of Waterloo

and

W.W. Wadge
Computer Science Department
University of Warwick

Abstract

Lucid is augmented by constructs allowing the use of local variables and defined functions. This is achieved while losing none of the aspects of Lucid as a formal system.

0. Introduction

We consider the addition to Lucid of features allowing local variables and defined functions. Such additions seem necessary if Lucid is to be a useful programming language.

The development of Lucid requires a different approach than the development of conventional programming languages. Lucid is first and foremost a formal logical system, with axioms and rules of inference, used for reasoning about particular sorts of formulas as assertions. Lucid programs are built up from such assertions in certain restricted ways. If the programming aspect of Lucid is to be developed, then the whole formal system must be developed also. A crucial feature of Lucid is that in a proof of a program, assertions derived at any stage have the same status, in the formal system, as the program itself. There is a smooth continuum from straightforward mathematics to Lucid programs. Thus if new constructs are developed for the programming language, one must consider the more general use of these constructs in proofs where they may be used not for defining computations but for stating more general assertions. We can add any assertion derived from a program to the program itself without changing its meaning. What if such assertions, which need not look at all like pieces of program themselves, are added within our new constructs?

Therefore, what must be developed is the formal system and, as before, special syntactic subclasses of formulas will correspond to our improved Lucid programs. So, in what follows, we consider the new constructs as new types of assertions, as might occur in proofs. We consider such questions as: what are the free and bound variables in such assertions, what equiva-

lence preserving manipulations can we perform on such assertions, and how do we substitute terms for variables in such assertions (a common operation in logical systems). We also indicate the extra syntactic restrictions that must be obeyed if these assertions are to be used as parts of programs.

There will be no formal semantics in this paper of the type given in [1]; the presentation will be informal. However the equivalence preserving manipulation rules we present should be sufficient to convey the meaning of constructs at an intuitive level, and should enable programs to be written and proofs carried out.

1 Clauses

Lucid has "constructs" for structuring programs analogous to the blocks, while loops and procedure declarations of Algol-like imperative languages. In the more general framework of the formal theory, clauses are collections of assertions considered as single assertions. They are used in programs to define variables (data variables or function variables) by means of locally defined variables. The "scope rules" will turn out to be essentially the same as those for Algol.

1.1 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 <expression> using <variable list>
    <set of assertions>
end.

The <expression> is called 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; all other variables are local variables of the clause whether or not they occur within the clause. A produce clause is an assertion about the subject and about the globals of the clause whose free variables are the global variables together with the free variables of the subject expression. Inside the body, the special local variable "result" refers to the subject expression. (The variable "result" cannot be a global variable.) More

precisely, a produce clause asserts that there are values for the local variables which make all the assertions in the body true when "result" has the value of the subject expression.

A produce clause is used in a program as a pseudo-equation defining its subject and possibly some of its globals. In this case, the body of the clause is a sub-program which will contain definitions of "result" and of every local variable occurring in the body and also of those globals that are to be defined (these definitions may be by means of clauses).

All examples will be of programs, but generally arbitrary assertions could be used instead of definitions of variables, to give more general clauses.

Example 1

```
produce meanX using X, sqdev
    first SUM = first X
    first SUMSQ = first X^2
    next SUM = SUM + next X
    next SUMSQ = SUMSQ + next X^2
          I = 1 followed by I+1
    sqdev = (SUMSQ - SUM^2)/I
    result = SUM/I
end
```

Here "X" and "sqdev" are global variables, with "sqdev" being defined within the clause, and "result", "SUM", "SUMSQ" and "I" are local variables.

Example 2

```
produce minY using Y
    first result = first Y
    next result = if result < next Y then
                      result else next Y
end.
```

Here "Y" is global and "result" is local.

A local variable of a clause (other than "result") can be consistently renamed without changing the meaning of the clause, provided the new name is not a global and is different from all other locals occurring in the clause.

The produce-end parentheses of a produce clause can be removed (giving a set of assertions) by

(1) renaming all local variables so that they differ from all variables in the smallest enclosing clause that are globals of that clause or occur freely in that clause, and

(2) replacing all free occurrences of the variable "result" in the body by the subject expression.

For example, if the produce clause in example 1 occurs in another clause in which the variables "A", "B" and "C" are not globals nor occur freely, then the produce clause can be replaced by

```
first A = first X
first B = first X^2
next A = A + next X
next B = B + next X^2
   C = I followed by C+I
sqdev = (B - A^2)/C
meanX = A/C
```

This renaming rule implies that the "scope" of a local variable of a clause is just that clause.

1.2 Compute Clauses

Lucid has a type of clause, called a compute clause, similar to the produce clause, which is used to write nested loops. It is of the form

compute <expression> using <variable list>

<set of assertions>

end.

A compute clause asserts that there exist values for the local variables which make all the assertions in the body true when all the globals refer to their latest values and "result" refers to the latest value of the subject.*

In a compute clause in a program, the local variable "result" and any globals defined in the clause must be defined by quiescent expressions (see [1]). The other global variables are also taken to be quiescent. The compute clause replaces the begin-end construct described in [2]. In other words, its effect is, intuitively, to "freeze" the values of the global variables during 'computation' of the inner loop.

Example 3

```
compute next R using X, tolerance
   first app = X/2
   next app = (app + X/app)/2
   result = app as soon as X app^2 < tolerance
end.
```

* Formally, the compute clause is equivalent to the corresponding produce clause obtained by substituting latest X for all free occurrences of global variable X except those in the subject expression (see 2.6), replacing the subject expression E by latest E and the word compute by produce.

The effect of this loop is to define the value of "R" at time t+1 to be the square root of the value of "X" at time t, calculated to a precision determined by the value of "tolerance" at time t.

Example 4. Here we combine produce and compute clauses.

```
produce meanX using X, dev
   first SUM = first X
   first SUMSQ = first X^2
   next SUM = SUM + next X
   next SUMSQ = SUMSQ + next X^2
         I = 1 followed by I+1
   tolerance = SUM/(I×10,000)
   sqdev = (SUMSQ - SUM^2)/I
   compute dev using sqdev, tolerance
         first app = sqdev/2
         next app = (app + sqdev/app)/2
         result = app as soon as sqdev - app^2 < tolerance
   end
   result = SUM/I
end
```

Clearly, the produce clause gives the "running-mean" of "X", while also making "dev" be the "running-standard-deviation" of "X", calculated to a continually increasing precision.

1.3 Function Clauses

Function clauses are assertions about function variables. A function clause is of the form

function <function variable>(<variable list>) using <variable list>

<set of assertions>

end.

The function variable is called the function name; the variables within the parentheses are the formal parameters; together these constitute the function header. These variables in the header must all be distinct, and may not occur in the global list. The global variables are those in the global list together with the function name. These are also the free variables of the clause considered as an assertion.

A function clause is an assertion about the free variables. It asserts the truth, for all values of the formal parameters, of the produce clause obtained by replacing the word "function" by the word "produce" and by adding the formal parameter variables and the function name to the global list.

When used in a program, a function clause is a definition of the function name and possibly of some of the globals, and the body of the clause must be a subprogram defining the variable "result". The formal parameters must not be defined in the body. Since the function name is a global of the clause, the function name being defined can, of course, be used in the body, i.e. recursive functions are allowed.

Example 5

```
function min(X)
    first result = first X
    next result = if result < next X then result else next X
end
```

(Here the word "using" was dropped because the global list is empty.)

Example 6

```
function mean(X) using dev
    first SUM = first X
    first SUMSQ = first X^2
    next SUM = SUM + next X
    next SUMSQ = SUMSQ + next X^2
          I = 1 followed by I+1
    tolerance = SUM/(I×10,000)
    sqdev = (SUMSQ - SUM^2)/I
    compute dev using sqdev, tolerance
          first app = sqdev/2
          next app = (app + sqdev/app)^2
          result = app as soon as sqdev - app^2 < tolerance
    end
    result = SUM/I
end
```

Note that both "mean" and "min" are non-pointwise functions of their arguments, i.e. their values at time t do not just depend on the values of their arguments at time t.

1.4 Mapping Clauses

A mapping is a pointwise function. It is often useful to know that a function is pointwise, and so Lucid provides a fourth type of clause, called a mapping clause, for defining mappings. The general

form is

mapping <function variable>(<variable list> using <variable list>
<set of assertions>
end.

The function variable is called the mapping name of the clause.

A mapping clause asserts the truth, for all values of the formal parameters, of the compute clause obtained by replacing the word "mapping" by the word "compute" and by adding the formal parameters and the mapping name to the global ist.

When used in a program, the syntactic restrictions are those for a function clause, together with the constraint that the variable "result" and all globals defined within the clause must be defined by quiescent expressions. The globals and formal parameters are taken to be quiescent.

Example 7

```
mapping root(X) using tolerance
   first app = X/2
   next app = (app+X/app)/2
   result = app as soon as X - app^2 < tolerance
end.
```

The mapping "root" is pointwise in its argument and in the global "tolerance".

Example 8

Here we combine a function and a mapping.

```
function mean(X) using dev
   mapping root (X) using tolerance
         first app = X/2
         next app = (app + X/app)/2
         result = app as soon as X - app^2 < tolerance
   end
   first SUM = first X
   first SUMSQ = first X^2
   next SUM = SUM + next X
   next SUMSQ = SUMSQ + next X^2
   I = 1 followed by I+1
   tolerance = SUM/(I×10,000)
   dev = root ((SUMSQ - SUM^2)/I)
   result = SUM/I
end
```

2. Manipulation Rules

2.1 Adding global variables

There are three cases in which a variable can be added to the global list of a clause, without changing the meaning of the clause:

(a) when the variable does not occur freely within the body of the clause,

(b) when the clause is a produce or function clause and the variable to be added is not a formal parameter of the clause and does not occur freely in the body of the smallest enclosing clause, and is not a global or formal parameter of that clause,

(c) when the clause is a compute or mapping clause and the variable to be added is quiescent and is not a formal parameter of the clause and does not occur freely in the body of the smallest enclosing clause and is not a global or formal parameter of that clause. (In this case, for programs, the result will be a program only if the variable is defined to be quiescent.)

2.2 Removing global variables

A global variable may be removed from the global list of a clause if it could immediately be replaced according to the rules above (2.1).

2.3 Adding assertions

There are two cases in which assertions can be added to the set comprising the body of a clause;

(a) any assertion may be duplicated, giving two or more copies, and

(b) any assertion

<variable> = <expression>

can be added to a clause provided the variable is a local variable, other than a formal parameter, not occurring freely in the body of the clause.

2.4 Moving clauses

Lucid also provides rules for moving assertions in and out of clauses. These rules allow us to extend the techniques for nested proofs and program massage given in [1,2].

Each type of clause has its own "moving rules".

2.4.1 Produce and Function Clauses

Any assertion can be moved in or out provided all its free variables are global variables of the produce or function clause. If the assertion to be moved out also has "result" as a free variable, it can be moved out provided all free occurrences of "result" are replaced by the subject expression, in the case of a produce clause, or by the function header in the case of a function clause.

2.4.2 Compute and Mapping Clauses

The rule is the same except the assertion being moved must be a point-wise assertion.

A pointwise assertion is either

(a) a Lucid term whose free variables do not occur within the scope of a nonpointwise function. This is called a pointwise expression;

(b) a compute clause whose subject is a pointwise expression;

or

(c) a mapping clause.

2.5 Moves which change the moved assertion

A useful additional rule is that any compute clause with a pointwise subject expression can be moved into a compute or mapping clause, becoming the corresponding produce clause, and any produce clause with a pointwise subject expression can be moved out of a compute or mapping clause becoming the corresponding compute clause. In both cases, the free variables of the clause being moved must be global variables of the clause being moved into or out of, except that, when moving out, the variable "result" may occur, and is replaced by the subject expression.

Note: The corresponding transformation between mapping clauses and function clauses does not work.

2.6 Substitution

Clauses are examples of assertions in the Lucid formal system. One very basic operation that is used a great deal in formal systems is the substitution of a term for all free occurrences of a variable. We must define this notion for clauses.

We will assume that we are only concerned with substituting for data variables. In that case the variable to be substituted must occur (i) in the subject expression and/or (ii) in the global list. The result of substituting term t for free occurrences of variable X in the clause is then obtained by substituting t for all free occurrences of X either in the subject expression or in all assertions making up the body of the clause (in case (i) and (ii) respectively) and by replacing X in the global list by all the free variables of t.

We say the term t is free for X in the clause if the substitution of t for X would not result in any of the occurrences of local variables in the clause (including sub-clauses) becoming global variables of the clause. This notion plays a crucial role in most formal systems.

3. "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 output
    function min(X)
       first result = first X
       next result = if result < next X
             then result else next X
    end
    X = 3 followed by 1 followed by 2
    result = min(X^2)
end
```

We first rename the formal parameter of "min" so that it doesn't conflict with any variable used in the enclosing produce:

```
produce output
   function min(Y)
         first result = first Y
         next result = if result < next Y then result
                                          else next Y
   end
   X = 3 followed by 1 followed by 2
   result = min(X^2)
end
```

Now we can add an assertion to set up the correspondence between actual and formal parameters, and substitute for the actual parameters:

```
produce output
   function min(Y)
         first result = first Y
         next result = if result < next Y then result
                                          else next Y
   end
   X = 3 followed by 1 followed by 2
   Y = X^2
   result = min(Y)
end
```

The function clause can now be replaced by the corresponding produce clause:

```
produce output
   produce min(Y) using Y, min
         first result = first Y
         next result = if result < next Y then result
                                          else next Y
   end
   X = 3 followed by 1 followed by 2
   Y = X^2
   result = min(Y)
end
```

(The global function variable "min" of the produce clause could subsequently be dropped.)

We can now show that

Y = 9 followed by 1 followed by 4

in the outer produce clause. This can then be moved into the inner produce clause, because "Y" is a global variable of this produce clause. In the inner produce clause we can then get

result = 9 followed by 1 followed by 1.

This is then moved out of the inner produce clause, becoming

min(Y) = 9 followed by 1 followed by 1.

This gives us the value of "result", so that we eventually move out of the outer produce clause the assertion

output = 9 followed by 1 followed by 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. One such mechanism is the call by name rule as considered in Vuillemin [3]. To see that call by value does not work, consider

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

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

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

The mapping clause and the assertions about "X" and "Y" can be moved into the compute clause, since they are all pointwise assertions, and after simplifications we obtain

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

We can now drop "X" and "Y" as global variables of the compute clause, allowing us, without renaming the formal parameters to set up the actual/formal parameter correspondence and replace the mapping clause by a compute clause:

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

Taking "X = 0" into the inner compute clause, we then obtain "result = 0", which becomes "f(X,Y) = 0" when we move it out. Therefore, we get "result = 0" in the outer compute clause, and hence "f(X,Y) = 0" in the produce clause. This gives "result = 0" at this outer level, which moves out of the outermost (produce) clause to become "output = 0".

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

A more efficient mechanism than call by name is the "delay rule" of Vuillemin [3], and Lucid may be the first programming language that can actually use it.

It is also worth noting that implementing non-recursive 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.

4. More Exotic Features

4.1 Create Clauses

We have seen that the globals of a compute clause are "frozen" within the clause. We can "unfreeze" these variables, within the compute clause, by using a create clause, which turns out to be a dual construct of the compute clause in many respects. A create clause is of the form

create <expression> using <variable list>

<set of assertions>

end

The idea is that globals of the create clause may be quiescent outside the create clause but are unfrozen within it. In a program, the values of

the variable "result" and of any globals defined within the create clause need not be quiescent, but the value of the subject expression and of these globals, in the enclosing clause, will nevertheless be quiescent. (The unfrozen value of the subject expression is the value of "result" within the create clause.) Moreover, the globals defined outside the create clause must be quiescent in this enclosing clause.

Example 9

In this example, the loop defines the value of "R" at time t+1 to be the square root of the value of "X" at time t, calculated to a precision given by the minimum value of "tolerance" up until time t:

```
compute next R using X, tolerance
   first app = X/2
   next app = (app + X/app)/2
   create mintol using tolerance
         first result = first tolerance
         next result = if result < next tolerance
                           then result else next tolerance
   end
   result = app as soon as X - app^2 < mintol
end
```

4.2 Transform clauses

We also have the dual of mappings, which we call "transforms". A transform clause is of the form

```
transform <function variable>(<variable list>) using <variable list>
     <set of assertions>
end.
```

In a program, the arguments and globals of a transform should always be quiescent.

In contrast to mappings (and functions) the transform name is not a global of the transform clause, so recursive transforms are not allowed.

Example 10

```
compute next R using X, tolerance
    transform min(Y)
        first result = first Y
        next result = if result < next Y then result
                                          else next Y
    end
    first app = X/2
    next app = (app + X/app)/2
    result = app as soon as X - app^2 < min(tolerance)
end
```

4.3 Manipulating create and transform clauses

The manipulation rules for create and transform clauses are summarised below. In the following it is required that the free variables of an assertion being moved be global variables of the clause being moved into or out of. In addition, when moving out, free occurrences of the variable "result" are replaced by subject expressions or headers as appropriate.

a) Moving through other clauses

Produce and function clauses treat create and transform clauses like any other assertions (see 2.4.1).

Create and transform clauses are not pointwise assertions and may not pass unchanged into or out of compute and mapping clauses. However, a create clause can be moved out of a compute clause, becoming the corresponding produce clause.

b) Moving other clauses through create and transform clauses

Only pointwise assertions may move in or out unchanged.

However, produce or create clauses can be moved in, becoming the corresponding compute and produce clauses respectively. Conversely, compute and produce clauses can be moved out, becoming the corresponding produce and create clauses.

It is important to note that compute and mapping clauses can now not be considered as pointwise assertions if they contain create or transform clauses that are not contained within other compute or mapping clauses.

To illustrate these rules, we will manipulate Example 9. We first notice that "mintol" is quiescent in the compute, because it is defined by a create clause. Provided "mintol" does not occur freely in and is not a global of the clause containing the compute clause, we can add "mintol" to the global list.

```
compute next R using X, tolerance, mintol
   first app = X/2
   next app = (app + X/app)/2
   create mintol using tolerance
         first result = first tolerance
         next result = if result < next tolerance
                       then result else next tolerance
   end
   result = app as soon as X - app^2 < mintol
end.
```

Now we can move the create clause out of compute clause, becoming the corresponding produce clause:

```
produce mintol using tolerance
   first result = first tolerance
   next result = if result < next tolerance
                 then result else next tolerance
end
compute next R using X, tolerance, mintol
   first app = X/2
   next app = (app + X/app)/2
   result = app as soon as X - app^2 < mintol
end
```

Since "tolerance" now does not occur within the compute clause, it can be dropped from the global list of that clause. What we have obtained

describes the meaning of Example 9 without using create clauses. Thus the compute clause of Example 9, which is not a pointwise assertion, becomes a pointwise assertion by giving it a different global variable ("mintol" instead of "tolerance").

The usefulness of create and transform clauses is debatable, but the clauses have been included here because they arise naturally in the development of the theory.

5 Comment on the design of clauses

We have emphasised that clauses are designed as extensions to the formal system of Lucid, not just to the programming language. This distinction can be well illustrated by considering another construct which we investigated before settling on clauses.

A clause is a pseudo-equation, which in programming language terms means it is a sort of compound statement. It is possible to achieve the same effect in the programming language, by using a sort of compound expression. Instead of using

```
produce X using A,B
        .
        .
        .
   result = E
end
```

we could say

```
X = valof E using A,B where          .
                                     .
                                     .
                          end
```

Here valof E using A,B where

.
.
.

end

is a term. Thus construct is appealing in the context of the programming language, although it does have some drawbacks. (For example, globals could not be defined within the valof body, as they could in the produce body, without introducing complicated syntactic restrictions on the use of several valof terms within a single term.)

Its great drawback is that it doesn't fit well into the context of the Lucid formal system. The construct only seems to make sense if the body is a piece of program, i.e. definitions of variables. Using it in proofs could result in terms like

valof X+Y using Y where X > Y

end

To give meaning, a formal semantics, to such terms would involve talking about sets of possible values for the term, for example the possible values of the term above are all numbers greater than twice Y. This causes all sorts of problems in the formalism. We could insist on only giving meaning to whole equations using valof terms, such as

X = valof X+Y using Y where X > Y

end

which states that X is greater than twice Y. This weakens the case for valof terms as entities in their own right, and leads directly to the use of clauses instead.

Using clauses, from

```
produce X using Y
     X > Y
     result = X+Y
end
```

we immediately get

```
produce X using Y
      result > 2Y
end
```

from which we can remove the produce-end parentheses to give X > 2Y. We could also manipulate the valof term to give the same result. First we rename the local variable X of the valof body so that it differs from all variables occurring outside:

```
X = valof Z+Y using Y where Z > Y
                        end
```

Then we can remove the valof-end parentheses, letting the assertion Z > Y join the others in the program:

```
Z > Y
X = Z+Y.
```

From this we immediately get X > 2Y.

So we see that reasoning about valof terms is quite straightforward. Their only real drawback is a technical one--it is difficult to give them

a semantics. When this is satisfactorily overcome, they may well be added to Lucid. But the point we are making is that the difficulty is caused by the necessity for new constructs to fit into the more general setting of a formal system, not just a programming language.

6. References

[1] Ashcroft, E.A. and Wadge, W.W. "Lucid, a Formal system for writing and Proving Programs". SIAM J. on Computing, 5, 1976.

[2] Ashcroft, E.A. and Wadge, W.W. "Lucid, a Non-Procedural Language with Iteration". To appear in CACM.

[3] Vuillemin, J. "Correct and Optimal Implementations of Recursion in a simple Programming Language", 5th Annual ACM Symposium on Theory of Computing, Austin 1973.