Clauses: Scope structures and defined functions in Lucid

E. A. Ashcroft        W. W. Wadge

Abstract

In this 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. This is done using new constructs called clauses which 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 (or any anonymous ‘process’ or ‘actor’); and a “function” clause defines a sort of procedure with own variables (or a general kind of coroutine).

1 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 [5]). 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.

Objections to denotative languages are all based on the supposedly obvious fact that denotative programs cannot be executed, and that they give rise to no ‘activity’ (other than function calling); in other words, that they have no operational meaning. Thus iteration, for example, is supposedly impossible because it is an operational concept: assignment is necessary (to update loop variables) and transfer is needed (to go back and start another iteration, and to exit on completion). Furthermore, denotative languages must (so goes the reasoning) be inherently inefficient, because their implementations cannot take advantage of various operational shortcuts such as the moving of constant expressions outside loops or sharing of variables; and finally, it is argued, denotative languages must always be cumbersome because they lack the rich variety of constructs found in imperative languages: the for-loops, case statements, multilevel exits, own variables and so on, all of course invented to bring about a desired operational “effect”. The denotative programmer, on the other hand, is supposedly restricted to recursive function definitions, so that his languages can never be anything more than syntactically sugared variants of the λ-calculus.

The first version of the Lucid language (as described in [3], henceforth referred to as “Basic Lucid”), refuted some of these objections 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

*Published in ACM Principles of Programming Languages 17–22, 1978.
is indeed possible for denotative programs to have operational interpretations. The operational
terms 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. We define the new constructs, their mathematical semantics and the
class of programs, and give the (derived) manipulation rules for verification and transformation
of programs.

2 Clauses

Lucid has “constructs” for structuring programs analogous to the blocks, while-loops and proce-
dure 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.

2.1 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

\[
\text{produce } \langle \text{data term} \rangle \text{ using } \langle \text{variable list} \rangle \\
\langle \text{set of assertions} \rangle \\
\text{end}
\]

The \( \langle \text{data term} \rangle \) at the head of the clause is the subject of the clause, the \( \langle \text{variable list} \rangle \) is the
global list, and the \( \langle \text{set of assertions} \rangle \) 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.

Here is a simple example

\[
\text{program root using } A, B, C \\
D = B^2 - 4 \cdot A \cdot C \\
output = (-B + \sqrt{D})/(2 \cdot A) \\
\text{end}
\]

Here, \( A, B \) and \( C \) are the global variables and \( output \) and \( D \) are the local variables. If this
clause is 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. This is because 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 exist values for the local variables which make all the assertions in the body of the clause, is called a local variable. Here is a simple example

\[
\text{program root using } A, B, C \\
D = B^2 - 4 \cdot A \cdot C \\
output = (-B + \sqrt{D})/(2 \cdot A) \\
\text{end}
\]

Thus the meaning of a produce clause is independent of the choice of local variables. For
example, we could have used \( E \) instead of \( D \) in the clause above.

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 clauses can be nested.
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.

2.2 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

\[
\text{function } \langle \text{function term} \rangle (\langle \text{data variable list} \rangle) \text{ using } \langle \text{variable list} \rangle \langle \text{set of assertions} \rangle \text{ end}
\]

The function term is called the subject of the clause, and the data variables in the parentheses are called the formal parameters. As before the global variables are those in the global list but now 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.

Here is a typical function clause taken from a program:

\[
\text{function Root } (A, B, C) \\
D = B^2 - 4 \cdot A \cdot C \\
output = (-B + \sqrt{D})/(2 \cdot A) \text{ end}
\]

In this case the global list is empty and the word using is dropped.

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.

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 formal parameters, 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 the 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 [2]), result must be defined to be quiescent, e.g., by an expression of the form \( X \text{asa } P \).\(^1\)

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

\(^1\)In this paper we will abbreviate as soon as by asa and followed by by fby.
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.

3 An operational view

The mathematical semantics of clauses is given rigorously in [1]. It is simple and precise, even when stated formally, and is used to justify the rules of inference for reasoning about programs and to prove the 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, operational view of the semantics of clauses in programs, which extends the operational view of Basic Lucid programs in terms of loops, described in [3]. 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 inner loop. For example the following program statement

```plaintext
compute log10X using X
  I = 0 fby I + 1
  P = 1 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| < 0.00001
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| < 0.00001
end
```

has an inner compute clause which defines each value of C to be (an approximation to) the corresponding value of √A·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 values 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 \( \text{trans} \).

\[
\text{mapping } \text{trans}(L) \text{ using } \text{trans}, \text{dict} \\
\text{result} = \\
\quad \text{if } \text{null}(L) \\
\quad \text{then } \text{NIL} \\
\quad \text{else if } \neg \text{atom}(L) \\
\quad \text{then } \text{cons}(\text{trans}(\text{car}(L)), \text{trans}(\text{cdr}(L))) \\
\quad \text{else } L' \\
D = \text{dict fby cdr}(D) \\
E = \text{car}(D) \\
\text{entry} = E \text{ asa } \text{car}(E) \text{ eq } L \lor \text{null}(E) \\
L' = \text{if } \text{null}(\text{entry}) \text{ then } L \text{ else } \text{cdr}(\text{entry}) 
\end{align*}

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 \( \text{dict} \). Because we want the definition of \( \text{trans} \) to be recursive the function variable \( \text{trans} \) must appear on the global list.

Terms involving mappings can be thought of as giving rise to Algol-like “mapping calls”, 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 in addition to restricting the scope of definition, they also freeze their globals and parameters so that they 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 anonymous 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

\[
\text{produce } Y \text{ using } X \\
N = 1 \text{ fby } N + 1 \\
T = \text{first } X \text{ fby } T + \text{next } X \\
\text{output} = T/N 
\end{align*}

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 or 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

\[
\begin{align*}
N &= 1 \text{ fby } N + 1 \\
function \ Avg(X) \\
I &= 1 \text{ fby } I + 1 \\
T &= \text{first } X \text{ fby } T + \text{next } X \\
output &= T/I \\
end \\
Y &= \text{Avg}(N) \cdot \text{Avg}(N^2) \text{ asa } N \text{ eq 5}
\end{align*}
\]

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 \(\text{Avg}\) 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 MacQueen [4].

In fact, languages already exist which exhibit similar “features” to most of those found in
this operational interpretation of produce and function clauses, but the difference is that the
designers started with the operational view, selecting those features which conformed to a personal
and preconceived view of what processes and coroutines are. In Lucid on the other hand, any
operational interpretations must conform with the mathematical semantics of clauses, and this
in turn is simply the result of combining the Lucid view of iteration with the ability to define
functions and limit the scopes of variables. We can therefore claim, with some justification, that
our notions of “coroutine” and “process” are the “true” ones.

### 4 Program proofs by program manipulation

We have rules for transforming programs which can be rigorously justified using the formal se-
manitics. Using such transformations it is possible to reason about programs in a very natural
way.

The first two rules allow us to carry on normal reasoning within clause boundaries. The first
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 of function clause, and any such assertion which is pointwise\(^2\) can be moved in
or out of 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. We also need a rule which allows any function

\(^2\)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.
or mapping clause to be transformed to a produce or compute clause (respectively) by adding the formal parameters to the global list and by making any consistent substitution of terms for the formal parameter.

While these rules are natural and easy to use, we know of no small well-structured symmetric subset of them which are in some sense complete.

4.1 “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)
end
```

We first use the function transformation rule, with the formal parameter $X$ being replaced by $Z^2$, yielding

```
produce M
  produce Min(Z) using Z
    first output = first 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)
end
```

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

```
produce M
  produce Min(Z) using Z
    first output = first 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)
end
```
Then inside the produce we substitute \(3 \text{ fby } 1 \text{ fby } 2\) for every occurrence of \(Z\), perform some simple calculations and, after discarding unnecessary statements, we have

\[
\begin{align*}
\text{produce } M \\
\text{produce } \text{Min}(Z^2) \text{ using } Z \\
\quad \text{output } = 9 \text{ fby } 1 \text{ fby } 1 \\
\end{align*}
\]

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

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 \textit{call-by-name} rule as considered in Vuillemin [6]. To see that \textit{call by value} does not work, consider

\[
\begin{align*}
\text{produce } V \\
\quad \text{mapping } f(X,Y) \text{ using } f \\
\quad \quad \text{result } = \text{if } X \text{ eq } 0 \text{ then } 0 \text{ else } f(X-1,f(X,Y)) \\
\quad \quad \text{output } = f(1,0) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{produce } V \\
\quad \text{mapping } f(X,Y) \text{ using } f \\
\quad \quad \text{result } = \text{if } X \text{ eq } 0 \text{ then } 0 \text{ else } f(X-1,f(X,Y)) \\
\quad \quad \text{compute } f(1,0) \text{ using } f \\
\quad \quad \quad \text{result } = \text{if } 1 \text{ eq } 0 \text{ then } 0 \text{ else } f(1-1,f(1,0)) \\
\quad \quad \quad \text{output } = f(1,0) \\
\end{align*}
\]

Inside the compute clause we obtain

\[
\text{result } = f(0,f(1,0))
\]

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

\[
\begin{align*}
\text{produce } V \\
\quad \text{mapping } f(X,Y) \text{ using } f \\
\quad \quad \text{result } = \text{if } X \text{ eq } 0 \text{ then } 0 \text{ else } f(X-1,f(X,Y)) \\
\quad \quad f(1,0) = f(0,f(1,0)) \\
\quad \quad \text{output } = f(1,0) \\
\end{align*}
\]

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

\[
\text{result } = \text{if } 0 \text{ eq } 0 \text{ then } 0 \text{ else } f(0-1,f(1,0)).
\]
This simplifies to

\[ \text{result} = 0 \]

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

produce \( V \)
\[
\begin{align*}
    f(0, f(1, 0)) &= 0 \\
    f(1, 0) &= f(0, f(1, 0)) \\
    output &= f(1, 0)
\end{align*}
\]

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.

Finally, we will show that the manipulation rules can be used to settle the question of dynamic versus static binding of global variables of functions and mappings. Consider

\begin{verbatim}
compute U
    mapping f(X) using Y
    result = X + Y
end

Y = 1
compute result using f
    Y = 2
    result = f(3)
end
end
\end{verbatim}

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

\begin{verbatim}
compute V
    Y = 1
compute result using Y
    mapping f(X) using Y
    result = X + Y
end

Z = 2
result = f(3)
end
end
\end{verbatim}

The definition of \( Y \) can now be brought down into the mapping, and it is then straightforward to finally obtain “\( V = 4 \)”.
References


