Viscid, a vi-like screen editor written in Lucid*

William W. Wadge

Abstract

In this paper we describe a full screen editor program written entirely in Lucid, a functional dataflow language. Our goal was to verify that pure nonprocedural languages can be used to program ‘realistic’ applications, including ones involving interactive I/O.

The editor (which we call viscid) is similar to the vi editor of Berkeley UNIX(TM). Viscid is much simpler than vi but does handle basic features, such as inserting and deleting characters, opening and deleting lines, searching for patterns, and scrolling up and down. We give an overview of the viscid program together with a complete program for a simplified version. We discuss the techniques and methodology employed, and compare the Lucid program to a version written in Pascal.

The Lucid program is about half the size of the Pascal version, more modular, and if anything easier to write. These advantages are due primarily to Lucid’s treatment of variables (time varying values) which has more in common with spreadsheet systems than with conventional imperative languages.

We consider the remaining obstacles which must be overcome if nonprocedural languages are to be a practical alternative to C and Pascal.

1 Introduction

In this paper we describe a full screen editor program written entirely in Lucid, a functional dataflow language invented by the author and E. A. Ashcroft [1, 13]. The version or member of the family used (called pLucid) is based on the data types and ‘inner syntax’ of POP2 [2] and the ‘outer syntax’ of ISWIM [3]. The program runs on the pLucid interpreter written by A. A. Faustini, based on an earlier version by C. Ostrum.

The screen editor (which we call viscid) is similar to the vi editor of Berkeley UNIX [8]. Naturally, a complete implementation of vi is an enormous task in any language (and still out of the reach of functional programming). But our editor program does handle basic features, such as cursor control, inserting and deleting characters, opening and deleting lines, searching for patterns, and scrolling up and down.

Our first goal was to demonstrate that realistic applications can be programmed easily and elegantly in a functional language (or at least in Lucid). Of course, there already exist many large programs written in LISP and its descendants (including the new ‘lazy’ languages such as HOPE [3] or Miranda [11]). Unfortunately most of these programs involve typical LISP-ish applications. They use highly recursive algorithms for manipulating hierarchically structured data (for example, compilers for other functional languages). Furthermore, they almost always treat only static data, and deal with input/output only as an afterthought. Finally, those that do deal with dynamic activities and handle (say) interactive output usually resort to ‘dirty’ features which invoke side effects. As a result, there is very little published experimental evidence that the strictly applicative approach is suitable for a large class of extremely important practical applications. It should be clear that these comments also apply to logic programming as well.

We chose full screen editing as the application because it seems to exemplify those aspects (dynamic activity, interactive input/output) that many functional programmers shy away from.

Furthermore, even a simple screen editor represents a nontrivial programming task; the Pascal version of the program was 1000 lines long.

The entire viscid program (about 500 lines) is too large to reproduce in this paper; a complete listing can be found in an earlier report [12], and a simpler version is given (without any discussion) in an appendix of the Lucid book [13]. Here we will give only an overview of the program’s structure. We will, however, study a complete program for a similar but vastly simpler application. Finally, we will comment on some of the lessons learned concerning the style and methodology of functional programming.

2 Interactive Programming in Lucid

We begin by giving an extremely simple pLucid program. The program is only a few lines long but it illustrates the ideas behind our declarative approach to display-oriented interactive applications.

Our program maintains a simple one-digit counter displayed in the upper left-hand corner of the terminal display. The counter is increased by one every time the + key is struck, and decreased by one every time the − key is struck. Other keys are ignored. If the count is outside the range 0–9, the character # is displayed instead.

The basic idea underlying the pLucid solution is that the required program is actually a filter transforming an input stream into an output stream.

The input is simply the stream of characters corresponding to the user’s keystrokes—for example, +, −, +, +, −, . . . . The output is a stream of strings (sequences of characters) which, when sent to the display, produce the desired activity. Obviously, the output cannot simply consist of the digits 0, 1, 0, 1, 2, 1, . . . , corresponding to the successive numeric values of the counter. This would cause the digits to be written in a row across the screen from left to right, possibly in the middle of text being displayed when the program was invoked. Instead, the first output string is the ‘escape sequence’ (ANSI control sequence) which clears the screen and ‘homes’ the cursor. Thereafter, each item output is a string consisting of the appropriate digit (or #) followed by a backspace character. The backspace character ensures that each succeeding digit replaces its predecessor.

Here is the complete pLucid program.

```
CLEAR^HOME fby digit^''
where
  count = 0 fby
    case char of
      '+' : count+1;
      '-' : count-1;
      default : count;
    end;
  digit = if count >= 0 and count < 10
          then substr('0123456789',count+1,count+1)
          else '#'
        fi;
  CLEAR = '\33[2J';
  HOME = '\33[;H';
end
```

(In pLucid string constants use matching single quotes, and the C language conventions for non-printing characters. The symbol ^ denotes string concatenation. More detailed information can be found in the pLucid programming manual [4].

In reading a Lucid program, the most important thing to remember is that the statements are equations defining values which vary with time. The equation for count, for example, could be paraphrased as follows:

```
count = 0 fby
  case char of
    '+' : count+1;
    '-' : count-1;
    default : count;
  end;
```

In pLucid, equations are used to define values, and data values are computed on the fly. The above equation simply says that count will be increased by one every time the + key is struck and decreased by one every time the − key is struck. The default case leaves count unchanged.
The value of count is initially (i.e., at time 0) 0. Thereafter, the value that count will have at the next time instant depends on the current value of char. If that value is +, the next value of count is one plus the current value; if the value of char is -, it is one less than the current value; otherwise the next value of count is equal to the current value.

Any attempt to interpret the Lucid statements as assignments will result in hopeless confusion. It is more accurate, and far more helpful, to imagine that the variables count and digit are names attached to cells in a spreadsheet. The equation for digit can be thought of as specifying a formula to be placed in the cell labelled digit. Then whatever number is found in the count cell, the corresponding string in the digit cell is the current value of the formula if count>0 and ... ‘#’ fi.

The following is a complete Bourne shell script to run the counter program in the desired interactive mode:

```
stty -echo cbreak
luval -s -c -p counter.i
stty echo -cbreak
```

The command luval is the pLucid interpreter; it interprets an intermediate form of the program, in the file counter.i. The counter.i file is produced by the pLucid compiler. The p, c and s interpreter options suppress prompts and specify character by character input/output. The stty command turns off the UNIX echo and specifies character by character UNIX input/output. Notice that there is no post-processing of the Lucid program’s output, nor are there any side-effect producing ‘dirty’ features used in the program itself.

3 Livid, a line-oriented visual editor written in Lucid

Our next step is to present a complete solution to a simplified version of the screen editor: a line editor which maintains only one line of text. We simplify even further by allowing only 0 and 1 as characters, and by restricting the commands to allow only insertion and deletion of characters, and cursor movement.

More precisely, livid accepts only five characters, 0, 1, x, l and h. All others are ignored. The five valid characters are interpreted as follows:

- **0** Insert a 0 just before the character on which the cursor is resting; shift the part of the line which follows the insertion one space to the right.

- **1** Insert a 1 in the same way.

- **x** Delete the character under the cursor, and shift all characters to the right of the cursor one space to the left.

- **l** Move the cursor one space to the right.

- **h** Move the cursor one space to the left.

In each case it is assumed that the actions are valid; for example, when an h character is entered, it is assumed that the cursor is not already in the leftmost position. The response of livid to errors is not specified. To help avoid such problems, we specify that initially the displayed line consists of the characters 0 and 1 in that order.

The processing required by the livid program is much more complex than that required by the counter above. Nevertheless, it should be clear that livid is still just a filter transforming a stream of input characters into a stream of output characters.

The first string in the output stream clears the screen, displays the string 01, then homes the cursor. Thereafter the string that is output depends on the command character just input.
If the input character is an h, the backspace character is sent as output.

If the input character is an l, the character currently under the cursor is sent as output (this has the net effect of moving the cursor one position to the right).

If the input character is a 0 or a 1, we must output the character itself, followed by the entire sequence of characters that were under or to the right of the cursor, followed by enough backspaces to reposition the cursor. This will give the effect of part of the line moving to the right. (Some smarter terminals have escapes which perform this automatically.)

Finally, if the command character is an x, we must output the sequence of characters that were to the right of the cursor, followed by a blank character, followed by enough backspaces to reposition the cursor. This gives the effect of part of the line shifting to the left. (Again, some smart terminals can do this automatically.)

Here is the complete pLucid program to perform the indicated transformation.

```
outchars
where
  lhs = [] fby
    case command of
    'h':  tl(lhs);
    'l':  hd(rhs) :: lhs;
    '0':  '0' :: lhs;
    '1':  '1' :: lhs;
    'x':  lhs;
        default: error;
    end;
  end;

  rhs = ['0' '1'] fby
    case command of
    'h':  hd(lhs) :: rhs;
    'l':  tl(rhs);
    '0':  rhs;
    '1':  rhs;
    'x':  tl(rhs);
        default: error;
    end;
  end;

outchars = CLEAR^HOME^implode(lhs<>rhs)^HOME fby
    case command of
    'h':  '\b';
    'l':  hd(rhs);
    '0':  '0'^redrawrhs;
    '1':  '1'^redrawrhs;
    'x':  implode(tl(rhs))"' "backspaces(length(rhs));
        default: eod;
    end
    where
        redrawrhs = implode(rhs)^backspaces(length(rhs));
    end;

implode(m) = if m eq [] then '' else hd(m)^implode(tl(m)) fi;
backspaces(n) = substr('\b\b\b\b\b\b\b\b\b\b',1,n);
CLEAR = '\33[2J';
HOME = '\33[;H';
end
```
The `implode` function takes a list of strings and concatenates them into a single string. The definition of `backspaces` is naïve but simple.

It should be clear enough how the program works, although again the reader may wish to consult the `pLucid` Programming Manual. The output stream is denoted by the variable `outchars`. The variables `lhs` and `rhs` form the program's 'internal memory', like the variable `count` in the counter program. The variable `lhs` is a list of the characters to the left of the cursor, in reverse order. The variable `rhs` is a list of the characters under or to the right of the cursor, in left-to-right order. These two variables are easily defined by recurrence. For example, `lhs` is initially the empty list, and its value on the next step depends on the current command character. If this character is 0 or 1, the new value of `lhs` is the result of consing the command character onto the current value of `lhs`. If the command character is x, `lhs` is unchanged, but if the command character is h, the new value is the tail of the current one. And if the command character is l, the new value is the result of consing the head of the current value of `rhs` onto the current value of `lhs`.

`Livid` is invoked with a shell script similar to the one used for the counter program. It is instructive, though, to run the `livid` program in the normal mode, with strings quoted and escapes expanded. We find, for example, that if the input stream begins

```
'0' '1' 'h' 'x' 'h' '1' 'l' ...
```

then the output stream begins

```
'\33[2J\33[1H01\33[1H'
'001\b\b'
'101\b\b'
'\b'
'01 \b\b\b'
'\b'
'1001\b\b\b'
'\b'
'0'
'...
```

This experiment should have the effect of demystifying interactive computation (if it ever held any mystery in the first place). It makes it clear that the `livid` program is only transforming input data to output data, like any other program. The actual activity (characters being inserted, lines shifting) is generated by a physical device, namely the terminal's video display.

## 4 From Line Editor to Screen Editor

`Viscid` can be viewed as an elaborate extension of `livid`.

The most important way in which `viscid` extends `livid` is that `viscid` allows two-dimensional text. The `viscid` 'working text' is a sequence of lines, not just a single line. The cursor position determines a working line as well as position (column) in that line. At first sight it might seem that `pLucid`, which has no arrays, might have difficulty representing text. However, the characters in the file being edited are accessed sequentially, both by row and column, so that true random access is not required (this is often the case). After a little experimentation we found a simple and reasonably efficient representation of the working text which uses only lists, strings, and numbers, all of which are available in `pLucid`.

The contents of the working text and the position of the cursor are 'remembered' in four variables, `llines`, `ulines`, `cline` and `colno`. The variable `llines` is a list of lines below the cursor, each line being a `pLucid` string. The variable `ulines` is a list of lines above the cursor, in reverse order; the head of this list is the line above the cursor, the second component is the line above that one, and the last element of `ulines` is the first line of the text (unless the cursor is sitting on the first line, in which case `ulines` is empty). The variable `cline` is the line on which
the cursor is currently sitting, and \texttt{colno} is the number of the column containing the cursor. These variables all have simple inductive definitions; here are those of \texttt{ulines} and \texttt{clines}:

\begin{verbatim}
ulines = //list of lines above the cursor
    [] fby
    case cmd of
        "up":  tl(ulines);
        "down": cline :: ulines;
        "open": cline :: ulines;
        "newline": sfront(cline,colno-1)^insert :: ulines;
        "linedel": if llines ne [] then ulines else tl(ulines) fi;
        "endpat": julines;
        default: ulines;
    end;

llines = //list of lines below the cursor
    tl(initialtext) fby
    case cmd of
        "up":  cline :: llines;
        "down": tl(llines);
        "linedel": if llines ne [] then tl(llines) else [] fi;
        "endpat": jllines;
        default: llines;
    end;
\end{verbatim}

The second major way in which viscid extends livid is in the use of modes: viscid, like vi, has (at least) two different modes, namely \texttt{command} mode and \texttt{insert} mode. For example, the character \texttt{x} means \textit{delete the character under the cursor} when the editor is in command mode; but it means \textit{insert an x} when the editor is in insert mode. In viscid, insert mode is entered by using the \texttt{i} or \texttt{o} command, and exited by typing the escape key.

The viscid program handles modes by passing the stream of user keystrokes through a prefilter (a pLucid prefilter, not a UNIX prefilter). The prefilter imitates a simple finite-state machine whose internal state records the current mode. This ‘command decoder’ associates with each input character (i.e., with each user keystroke) a ‘command’ (the variable \texttt{cmd}) describing the nature of the information conveyed by the current input character. This command (which is a POP2 word) may correspond to one of those known to the user. For example, if \texttt{cmd} is the word \texttt{up}, it means that the cursor is to be moved up to the next line. The command decoder also produces pseudo-commands (such as \texttt{moreinsert}) which accompany characters sent in insert mode or in entering a pattern to be searched for.

Viscid does not restrict the user to files which can fit on the screen. Instead, it displays only those lines lying inside a ‘window’ which contains the line on which the cursor is resting. When a command repositions the cursor on a line outside the current window, the program brings the formerly ‘hidden’ line into view by scrolling or redrawing the screen. To do this, it is necessary to remember the line number of the line currently displayed at the top of the screen.

Finally, viscid checks that each command can be carried out; for example, if the command is \texttt{up}, it checks that the cursor is not already on the first line of text. It does this by passing the ‘raw’ commands through an error checking filter (again, this is a pLucid filter). This filter replaces infeasible commands by the internal command \texttt{error}, whose only effect is to cause the terminal to beep.

5 \textbf{The Structure of the Program}

The viscid program is a good example of the way in which the dataflow approach often suggests a natural decomposition of the problem into a number of well separated parts (Yourdon and
Constantine [14] give many examples of dataflow used as a structuring technique).

The input stream from the user’s terminal is passed through a pLucid pipeline consisting of three filters before it is sent as output to the terminal. The first filter is the command decoder, described above, which associates an internal command with each input character.

The second filter in the pipeline is the character generator. It generates the stream of text and escape sequences to be sent to the terminal display. The character generator sometimes produces entire lists of strings. This simplifies the programming and avoids inefficiencies involved in concatenating long strings. The lists of strings in this stream are ‘unravelled’ by the third filter, readout. The readout filter receives as its input a stream of strings and lists of strings and lists of lists of strings, etc. It produces as its output a stream of the strings occurring anywhere within its input, in the order they were received. (Note that in general, readout produces output faster than it receives input).

The character generator is by far the most elaborate. It is defined by a pipeline of two ‘internal’ filters, namely errcheck and outchars. The errcheck filter (described earlier) weeds out commands that cannot be executed. The second filter in the pipeline, outchars, generates the bundles of characters sent to the screen. The enclosing character-generating filter has a large internal memory to which both its subfilters have access.

This structuring of a dataflow program in terms of filters is analogous to the structuring of a conventional program using procedures. The dataflow version is more flexible, because filters are continuously operating transformations with internal memory (somewhat like coroutines).

In the course of developing the viscid program we used another, slightly different structuring of the program. PLucid has a simple include feature like that of C. Once the program reached a certain size, it became convenient to break it up into a number of smaller files. In dividing the program up, it was only natural to group the definitions of related variables—for example, the variables which maintain the state of the working text. In each grouping, some of the variables defined are used in other parts of the program, whereas others are used only within the group. These groupings constitute an informal module structure, with each module hiding the details of the definitions of its variables. Of course, this structuring is not really supported by pLucid. There is no separate compilation of definition files, and all variables defined in a given file could in principle be used elsewhere. But even with these limitations the module structure proved extremely useful.

Here is the main program.

```plaintext
changen(cdecode(C))
where
cdecode(C) = cmd //turn raw stream of chars into stream of commands
where
   include "cdecoder";
end;
changen(rawcmd) = readout(chars) //generate the control chars from the commands
where
   include "errcheck"; // define the sanitized cmd in terms of rawcmd
   include "outchars"; // generate chars to update the screen
   include "window"; // keep track of window position
   include "workingtext"; // maintains the current state of working text
   include "pattern"; // searching for a pattern
   include "jump"; // updating the screen after a cursor jump
   include "escseqs"; // ANSI control sequences for the
   include "strings"; // useful string operations
   include "lists"; // useful list operations
   include "gather"; // functions for turning lists into streams
      // and vice versa
end;
end;
```
Recall that pLucid ‘functions’ are actually filters; an expression like \texttt{chargen(cdecode(C))} therefore denotes a pipeline in which stream \texttt{C} is passed through the filters \texttt{cdecode} and \texttt{chargen}, in that order. In UNIX notation (not supported by pLucid), we could write \texttt{cdecode < C | chargen}. The pipelines and filters referred to earlier are pLucid ones, expressed in pLucid notation.

6 The Use of Variables

We have already seen that Lucid programmers can define variables—values which change with time. We would therefore expect that the Lucid screen editor program would in many respects resemble a program written in an imperative language like Pascal. We wrote such a program and found that this was in fact the case. Many of the variables defined in the Lucid program had analogs in the Pascal version. It was even possible, in the Pascal version, to imitate (to a limited degree) the dataflow structure of the Lucid version. For example, there is a procedure \texttt{nextcmd} which when called gets the next input character and its associated command.

Nevertheless a comparison of the Lucid version of viscid with the Pascal version reveals profound differences, mainly involving the treatment of variables.

In an imperative language, a variable is a passive object (essentially a container) which changes only as a result of external influences. In Lucid, however, a variable can be regarded as a self-sufficient entity whose values are generated or maintained by an autonomous device. The Lucid programmer must specify in one place (in the definition of the variable) the rule(s) according to which the variable’s value changes from one timepoint to the next.

In the viscid program, for example, there are a number of definitions which involve a \texttt{case} statement depending on \texttt{cmd}. Each of them describes separately the way in which a particular variable changes in response to the current editor command. Each of these separate definitions can be placed (as we have seen) in a separate ‘module’. In the Pascal program, there is one big \texttt{case} statement, each arm of which updates several variables.

In the Pascal version it was not possible to isolate, in a single place, every command which might affect a particular variable. We did, of course, ensure that all accesses or updates are performed using only a limited set of ‘interface’ procedures. We then gathered together all the procedures which affect a given group of variables. But there was no way to gather together all the \texttt{calls} to these procedures. In particular, calls to the procedures which maintain the working text were scattered all over the big \texttt{case} statement which executed the internal commands. There is no simple way to examine the Pascal program and determine the way in which (say) the cursor column number changes from one moment to the next.

The second difference (between Lucid and Pascal) is that Lucid is a lazy language—values are not computed unless they are actually needed. As a result, many of the definitions in the program are more like procedure calls, in that the indicated computations are carried out only when the values defined are needed. For example, the program contains a definition of the variable \texttt{plineno}. The value of \texttt{plineno} at any time is the line number of the first occurrence in the text (after the cursor position) of the current value of the string \texttt{pattern}. The definition specifies a value for \texttt{plineno} with each step in the main iteration. This does not mean, however, that every user keystroke invokes a search of the text. Only those values of \texttt{plineno} will be demanded which correspond to a request by the user to find a pattern.

There are many examples of definitions like this, of values which will be needed only intermittently. The pLucid interpreter automatically computes only those values which are required. The programmer does not have to explicitly specify the conditions under which a particular computation must be performed.

Finally, the definitional approach to the history of variables allows the programmer to use Lucid’s special temporal operators to access values of variables at other than the current time, when appropriate. For example, some commands can cause the text window to move long distances and make redrawing the screen necessary. It is therefore necessary to ask whether the next value of \texttt{textlineno} will be greater than the current value of \texttt{wbottom} (the line number of the bottom of the screen window). This test can be written as \texttt{(next textlineno) > wbottom}. 
7 PLucid as a Systems Programming Language

Since nonprocedural languages are based on formal mathematical systems, it is hardly surprising
that they are well-suited to mathematical applications. There are, for example, many elegant
examples of programs which producing infinite lists of primes, or the decimal digits of e. By
the same token, it is only natural to expect nonprocedural languages to be less well suited to
nonmathematical applications, in particular for systems programming, which has the reputation
of being inherently complex and conceptually dirty.

In our experience, however, pLucid proved remarkably well suited to at least one ‘systems’
application, namely screen editing. Part of the credit, of course, goes not to the language itself,
but to the implementation. The pLucid interpreter was designed to work easily with UNIX, and
has excellent features for controlling input/output and interfacing with the operating system. We
should mention that none of these features were added after the fact, to make the screen editor
easier to write. The pLucid interpreter was completed in January of 1983, several weeks before
we took up the screen editor ‘challenge’ (see below).

The entire pLucid viscid program is only 500 lines of source. Of these 500 lines about 200
define general-purpose utilities (e.g., for matching strings). The logic for the editor itself is around
300 source lines, about 50 definitions. By contrast, the Pascal version was more that 1000 lines of
source, and (as we mentioned earlier) somewhat less modular in its structure.

We had no particular difficulties developing the program. We proceeded incrementally, starting
with a program like livid and expanding it to deal with two-dimensional text and then adding one
command at a time. Windowing and pattern searching were the last to be added. Debugging was
fairly straightforward. During the development phase we altered the top line of the program to
an expression such as \[%ulines,cline,llines,colno\%\], which gives snapshots of the contents of
the most important variables. It was not necessary to embed debugging code inside the program
itself (this technique is described in more detail in the Lucid book [13]).

Although pLucid was adequate for viscid, we uncovered some weaknesses which could cause
trouble with more ambitious projects.

The present interpreter has only one output. An editor, however, has two outputs: the charac-
ters sent to control the display, and the final state of the text. We were forced to use the pLucid
system interface in a dirty way, to cause the final state of the text to be written back to the disk.
This is the only use of side effects in the whole program.

A compile-time typechecker could have spared us most of our debugging effort. It would not
be difficult to adapt the type discipline of ML [5] to pLucid. Almost all of the viscid program
would pass such a typechecker unchanged (the readout filter would need reworking). The pLucid
interpreter has no such static typechecking. It does not even check that functions are used with the
correct number of actual parameters. Fortunately, it does have excellent runtime error detection
and reporting.

PLucid’s facilities for breaking up large programs (include statements) is crude and was barely
adequate for the viscid program. Adding separately compilable modules would not, by itself, be
particularly difficult. The real challenge is to combine information-hiding modules with a strong
type checker, providing abstract data types. This topic is under investigation.

8 Conclusions

The viscid effort certainly constitutes further evidence that nontrivial and nonmathematical ap-
plications can be written in a pure functional language. Furthermore, it conclusively refutes any
claims that functional languages are suitable for static applications only, and that nothing can
be made to ‘happen’ or ‘change’ without commands and assignment statements. More signifi-
cantly, we have shown that a functional language can be a practical alternative, at least from the
programming point of view.

We will not claim, on the basis of program sizes, that Lucid programmers will be twice as
productive as Pascal programmers—although it is interesting that Peyton Jones [7] recorded the
same two-to-one ratio between imperative and functional programming. The Lucid program was
simpler than the imperative one, but not drastically simpler. Although nonprocedural languages
have (we believe) significant advantages, these advantages are not necessarily quantifiable in terms
of lines of code, nor are they all necessarily apparent in a single programming project of modest
size. The real point is that the Lucid programming was of comparable complexity. Arguably,
it was easier; but in any case, it was not drastically more difficult to use a pure nonprocedural
language. This is the real significance of our experiment.

It would be instructive to compare the viscid program with a version written in another func-
tional language, especially one based, like the SASL–KRC–Miranda family, on LISP and the
\( \lambda \)-calculus. As it happens, this problem (to write a functional screen editor) was issued as a
challenge by Graham Hill of GEC (UK) at a functional-programming workshop in London in
early 1983. Nevertheless, to our knowledge, none of the better known proponents of functional
programming has ever publicly accepted this challenge.

This is unfortunate, because in principle there is nothing to prevent a viscid program being
written in, say, KRC. As Turner points out \[11\], it has long been known that stream input/output
can be modelled in a lazy language which allows infinite lists. Such programs may well exist; but
even so, we know of no published account of their structure, their capabilities and performance,
and (most important of all) the extent to which side effects are employed.

We do have first-hand information about one editor written in a \( \lambda \)-calculus language, although
admittedly it may not be representative. In 1983 Andrew Minter, then a research student at
the University of Warwick, did produce a simplified viscid program, without window scrolling
and pattern search. He wrote it in CFL, a fairly standard \( \lambda \)-calculus language he developed and
implemented himself. The program was not particularly large, but it was rather opaque, and
the programming was, in Minter’s words, diabolical. The problem seems to be that conventional
LISP-like functional languages normally simulate iteration with tail recursion. In this approach
the variables which change as the iteration progresses become parameters of the recursively defined
function, and this is highly inconvenient. The programmer must remember the significance of (say)
the fourth parameter. Adding or removing a variable means changing the entire definition of the
function, and learning the new ‘numbering’ of the parameters. We doubt whether it would be
practical to implement full viscid (with dozens of variables) using this method. But there may be
alternative approaches, involving perhaps higher order functions.

9 Future Work

The work reported here represents only a modest step in the development of a practical nonproce-
dural approach to programming. A great deal of work remains to be done before a language like
pLucid is a real alternative to Pascal or C.

Viscid is a very primitive editor, compared to vi or emacs. We have shown that basic features
can be implemented using pLucid, but what about more complex ones, such keyboard macros
or pattern substitution? Fortunately, at least some of these advanced features do not present
any real problems. The author and at least one colleague (A. A. Faustini) have used the viscid
program as the basis for several term projects in graduate and advanced undergraduate courses.
These projects involved extending the editor to handle extra features, and none of the features
were particularly difficult to program. The viscid program has also been used in similar projects
as the basis for other interactive programs, for example, for a simple interactive game. The design
presented here (command decoder, error filter, character generator) has proved sound and flexible.

Even the most elaborate screen editor is, by some standards, a relatively straightforward piece
of software. Vi itself is hardly the most complex part of UNIX. Graham Hill, when he gave his
challenge, also suggested two or three even more ambitious projects, for example implementing
one of the standard communication protocols. The ultimate challenge, though, is to write an
entire operating system in a pure functional language. This last task is probably not possible in
pLucid, which is a first-order language. The more conventional LISP-based functional languages
have higher-order functions, but not streams, at least not in a natural way. Writing an operating
system would seem to require a language based on a higher-order temporal logic. This is another topic currently under investigation.

Of course, even the most sophisticated editor is useless if it performs poorly—runs slowly and/or consumes a lot of space. Performance has always been the biggest problem with nonprocedural languages. The viscid project did not really address the performance issue. Our goal was to show only that the programming effort was not unreasonable, regardless (almost) of how the resulting program performed (Hill’s challenge did not specify any performance level). As it happened, the earlier versions ran fast enough to be usable. In particular, during insertion in the middle of line, they could keep up with a reasonable typing speed. The fully loaded version, however, is very sluggish, and can handle only small files (a few lines long).

At the moment there is no pLucid implementation on which viscid is useful. Some recent work indicates that a pLucid compiler could run programs as much as an order of magnitude faster. This speedup would be enough to make viscid usable, but at present the compilable subset of pLucid is too small to include the viscid program. Of course, a special-purpose Lucid dataflow machine would also solve the problem, but at present this is still yet another topic under investigation.

Finally, even the most elegant and efficient of programs is useless if it is incorrect. The viscid program seems to be correct, but only because it has been extensively tested. The program was developed using conventional (or rather, naïve) techniques. We did not even use a formal specification. This is somewhat ironic, since Lucid was originally a language/system designed to assist formal correctness.

The problem here is not with the language. In a sense, the viscid program already is a specification: the definition of each variable specifies the sequence of values taken. As a specification, however, it is not very clear, because the equations have been written so as to be efficiently executable (here efficient means runs fast enough to be tested).

It should not be hard to produce a really inefficient version of viscid in which everything is sacrificed for clarity. This exceptionally clear version could be taken as the formal specification, so that proving correctness reduces to proving the equivalence of the two versions. One way to do this is to transform one version into another. In the Lucid book [13], we give some transformation rules and some examples of their use. The viscid transformation, however, would probably be substantially more difficult.

References


