An Eductive Interpreter for the Language pLucid∗

A. A. Faustini  W. W. Wadge

Abstract

We describe an interpreter for pLucid, a member of the Lucid family of functional dataflow languages. In appearance, pLucid is similar to Landin’s ISWIM, except that individual variables and expressions denote streams (infinite sequences of data items), and function variables denote filters (stream-to-stream transformations). The actual data objects in pLucid (the components of streams) are those of POP2: numbers, strings, words and lists. The ‘inner syntax’ (infix operations, conventions for denoting constants) are those of POP2 as well.

The interpreter (which was written in C) is eductive: it uses a tagged demand-driven scheme. Demands for values in the output stream generate demands for values of other variables internal to the program. These demands, and the values returned in response, are tagged according to “time” (sequence index) and place (node in the tree of function calls). Once computed, values are stored in an associative memory (the “warehouse”) in case they are demanded again later in the computation. The warehouse is periodically cleaned out using a heuristic called the “retirement plan”. The heuristic is not perfect, but it does not have to be: in an eductive computation, the program is not altered, as in reduction. If discarded values are needed again, they can be recomputed.

The pLucid interpreter performs extensive runtime checks and error messages quote the source line containing the offended operator. A special end-of-data object permits a very simple treatment of finite (terminating) input and output. Of special interest is its interface to UNIX¹, which allows any system command to be used as a filter inside a pLucid program.

The interpreter performs well enough for nontrivial programs to be developed and tested. These include (simple versions of) a text formatter, a distributed airline reservation system, and a full screen editor.

1 The Language pLucid

pLucid is a member of the Lucid family of functional dataflow languages [2]. The particular members of this family are determined by the primitive or built-in data types and operations. The data types and operations of pLucid are (almost) exactly those of POP2 [4], a LISP-like AI language popular in the UK. The basic data types of POP2 and pLucid are numbers, strings, words (analogous to LISP atoms) and arbitrary finite lists thereof. pLucid also adopts POP2 notation for these operations, so that (for example) <> is the list-append operator and [dog 3 ‘bob’] is a list constant denoting a list whose three elements are a word, a number and a string.

Here is a simple pLucid program:

\[
\text{rev}(m) \\
\text{where} \\
\text{rev}(x) = \text{if } x \text{ eq } [ ] \text{ then } [ ] \text{ else rev(tl(x)) <> [ hd(x) %]}
\]

This program inputs lists and outputs them in reverse order.

¹UNIX is a trademark of A T & T Bell Labs.
The ‘outer syntax’ of pLucid is essentially that of Landin’s (1966) ISWIM. In pLucid all where clauses are recursive, so that pLucid’s where corresponds to Landin’s whererec. pLucid is a first-order language: the programmer cannot use or define higher-order functions, that is, functions which accept other functions as arguments and/or return functions as results. Without higher-order functions the λ-notation is of limited use and is therefore not supported in Lucid.

pLucid is therefore a rather restricted and conventional-looking functional language whose main features have been plagiarized from ISWIM and POP2. At the same time, however, pLucid compensates for inadequacies and lack of originality in one very important way: it allows iteration. In pLucid, programmers can express iterative (or, more generally, datatow) algorithms in a very natural manner, without using tail recursion or infinite lists or similar clumsy devices.

Iteration is built right in to pLucid (and all members of the Lucid family). In Lucid, individual variables and expressions denote entire streams of ordinary data objects. Furthermore, programmers are encouraged to think operationally, in terms of streams and the flow of data. In the same way, function variables in pLucid do not denote simple functions which map single data objects to single data objects; instead, rather, they denote filters which transform streams to streams. Programmers are encouraged to think operationally about functions as well: to imagine them as continuously operating devices which are endlessly accepting data on an input line, performing internal calculations, and sending data on an output line (in the spirit of Kahn and MacQueen [5]).

Here is a simple pLucid program which produces running statistics on the numbers in its input stream. It produces a list of length three for every number input. The first item in the list is the minimum of the numbers input so far, the last is the largest of those input so far, and the second item in the list is the average of all those input so far.

```pLucid
[% min, tot(x)/n, max %]
where
  min = x fby if min > next x then next x else min fi;
  max = x fby if max < next x then next x else max fi;
  tot(a) = s where
    s = a fby s + next a;
  end
  n = 1 fby n + 1;
end
```

In this program, the variable x has no definition. It is therefore taken as the input to the program. If the stream of values input to the program begins 3, 5, 8, 4, ..., then the program, considered as an expression, has a stream value which begins [3 3 3], [3 4 5], [3 5.333 8], [3 5 8], .... There are the values which the program will output (see §5 for more on the pLucid input/output convention).

The values of the individual variables defined in the where clause (namely, max, min, and n) are also streams, and the values of max and min depend on the value of x. If the input is as described, the value of max begins 3, 3, 3, ... and that of min begins 3, 5, 8, 8, .... The value of n is always the stream 1, 2, 3, 4, 5, .... Sometimes individual variables are better thought of as time-varying objects, which are first initialized and then repeatedly updated. The sequences defined are then the histories of values of these variables.

The function tot is defined in the where clause as a filter which produces a running total of its argument. If an input (argument) stream of tot begins 3, 5, 8, 4, ..., then the corresponding output (result) stream begins 3, 8, 16, 20, ....

Of course the ‘official’ semantics of pLucid is purely static, and uses infinite sequences and (as with ISWIM itself) least fixed points. It is important, however, to understand that the operational views described above complement the denotational semantics—they do not contradict it. The dynamic interpretation is not ‘improper’ or ‘non-abstract’. In fact, it is almost impossible to read and write pLucid programs without the assistance of the operational interpretations of variables and functions. It is very wrong to think that Lucid is simply a static functional language with
facilities for manipulating infinite sequences. Instead, it must be understood as an extended functional language with facilities for iteration and dataflow.

We have argued elsewhere that Lucid is best thought of as a functional language based on intensional logic.

It could (and has) been argued that Lucid is not a functional language at all—because it is first order, and does not allow functions to be treated as data objects. On the other hand, Lucid’s only ‘constructs’ are those of ISWIM, for defining and applying functions. In particular, Lucid’s iteration/dataflow features are presented to the user as extra predefined functions (first, next, fby, etc.)

This whole discussion could be avoided by adding higher-order functions to Lucid. There is no reason in principle why this cannot be done, but there are many subtle semantic difficulties associated with higher-order intensional logic. Furthermore, we do not yet know how to extend the ‘place-parameter’ scheme (described below) to handle higher-order programs.

2 Eduction

The first attempts at implementing Lucid involved translating programs into a conventional imperative language (perhaps with a coroutine feature) or into dataflow (producer/consumer) networks. This approach is naturally suggested by the operational or dynamic interpretation described earlier.

Unfortunately neither of these methods can ever form the basis for a completely general implementation. The problem is that in general we cannot expect Lucid ‘time’ (the sequence index) to correspond exactly to real computation time. A Lucid filter may need to consume its input items in an order different from that given by the formal semantics, and may not be capable of producing its output values in the order expected. Even worse, it may not require all its input values, and may not be capable of producing a genuine value for every component in the stream (because the required computations do not terminate). Even apparently harmless expressions can give rise to such anachronistic or intermittent streams. For example, the expression

\[
\text{if next } i = 0 \text{ then } y \text{ else } c/i fi;
\]

will consume the values of \( i \) out of index order, may only require a few of the values of \( y \), and may diverge for certain index values.

The failure of the early coroutine/dataflow approach to implementing Lucid forced researchers to abandon any attempt to interpret the Lucid (sequence-index) time as real computation time. Two graduate students (T. Cargill at the University of Waterloo and M. D. May at the University of Warwick) simultaneously and independently devised a completely different technique, based directly on the formal semantics of Lucid. This technique was extended by C. B. Ostrum to handle nested iteration and user-defined functions and formed the basis for his Luthid interpreter [6]. (Luthid was a more LISP-like precursor of pLucid, and the pLucid interpreter is based on Ostrum’s original Luthid interpreter.)

This technique was briefly described in [2] and is now called eduction. Eduction is an extremely simple concept and is easily explained with an example. Consider the program

\[
j \text{ where} \\
i = 1 \text{ fby } i + 1; \\
j = 1 \text{ fby } j + 2 \times i + 1; \\
\]

which outputs the stream 1, 4, 9, 16, 25, ... of squares of natural numbers. The first number which should be produced is the value of the whole program at (Lucid) time 0, and this is the value of the value of the variable \( j \) at time 0. The variable \( j \) is defined by an expression involving fby. The formal semantics tells us that the result of applying fby to streams \((x_0, x_1, x_2, \ldots)\) and
\( \langle y_0, y_1, y_2, \ldots \rangle \) is the stream \( \langle x_0, y_0, y_1, y_2, \ldots \rangle \). This means that the value of \( \mathbf{fby} \ j + 2 * i + 1 \) at time 0 is the value of 1 at time 0, which is of course 1.

The next value we need is that of the program at time 1, which is in turn that of \( j \) at time 1. We again look up the definition of \( j \) and this time the formal semantics tells us that the value of \( \mathbf{fby} \ j + 2 * i + 1 \) at time 1 is the value of \( j + 2 * i + 1 \) at time 0. Since data operations (like addition) work componentwise, the value of \( j + 2 * i + 1 \) at time 0 is the sum of the values of \( j, 2 * i \) and 1 at time 0. The first has already been calculated (it is 1). The second is the product of the value of \( i \) at time 0 (which is 1) and the value of 2 at time 0 (which is 2). The second summand is therefore (at time 1) 2. Finally, the value of 1 at time 1 is 1, and the final result is therefore 4.

The basic principle of the eduction approach should now be apparent. The entire computation is driven by the attempts to compute in turn the time-0, time-1, time-2, ... values of the output. In computing the value of the output at some particular time, the ‘educer’ is led to compute values of various program variables at various different times. When the value of a variable \( v \) at time \( t \) is required, the educer consults the relevant definition of the variable. It uses the formal semantics of the operations occurring in the definition and computes the required value from the computed values of the operands. In general it will have to compute the values of operands at times other than \( t \); and it often requires values of the same operand at two or more different times.

For example, if the educer needs the value of \( A \) at time 8 and \( A \) is defined as \( B + C \), it computes \( B \) at time 8 and \( C \) at time 8 and returns their sum. If \( A \) is defined as \( \mathbf{fby} \ B \), it computes the value of \( B \) at time 7 and returns it. If the defining expression is \( \mathbf{next} \ B \), it computes the value of \( B \) at time 9. If the expression is \( \mathbf{if} \ P \ \mathbf{then} \ X \ \mathbf{else} \ Y \ \mathbf{fi} \), then the educer first computes the value of \( P \) at time 8. If this value is \( \mathbf{true} \), it computes and returns the value of \( X \) at time 8; otherwise it computes and returns the value of \( Y \) at time 8. Other operations are handled in a similar fashion (function ‘calls’ will be discussed later).

Eduction can be understood as a form of dataflow in which there is a two-way traffic in the communication lines. In one direction, data flows, from producers to consumers, in the usual way. In the other direction, demands are sent, from the consumers upstream to the producers. Both the demands sent upstream and the data sent downstream are tagged (in the simplest case, with a sequence index, or “time parameter”). The real-time order in which tagged data items are sent down communication lines is irrelevant. There are no FIFO queues on the input lines of filters. The kind of storage needed for an eductive configuration is very different, and will be discussed later.

Eduction is therefore simply tagged, demand-driven dataflow. The tagging solves the problem of anachronistic streams, and the lazy (demand-driven) evaluation permits intermittent streams. Of course, it could be argued that “demand-driven” dataflow is not dataflow at all. To many people, the defining property of dataflow is that operations are performed as soon as the operands are available. We could certainly argue about this point; but it would be an argument about words, and not about computation, rather like the argument about Lucid being a “functional” language. Instead, we have (following Ashcroft and Wadge) adopted the word “eduction”.

3 Memory in the Interpreter

One of the most suprising properties of eduction is that it does not require any static memory at all. The program is not changed at all during an eductive computation. As a result, values that have been computed but not saved can be recomputed (from scratch) if they are needed again. Of course, a memoryless educer will be hopelessly inefficient. For example, if \( i \) is defined as

\[ i = 0 \mathbf{fby} \ i + 1; \]

then the computation of the time \( t \) value of \( i \) involves computing the values of \( i \) at times \( t-1, t-2, t-3, \ldots \); right down to time 0. The simple pLucid squares generator given in §1 is apparently linear in complexity, but without memory, the interpreter has to perform \( O(N^3) \) operations to output the first \( N \) squares.
The interpreter avoids endless recomputations by storing computed values in a large “warehouse”. Everytime a new demand is generated, the interpreter first checks the warehouse to see if the value is not already available. Values in the warehouse are labeled by (1) the variable whose value is stored and (2) the tag identifying the particular value in question. Items in the warehouse are accessed by their labels; it is an associative memory. Accesses are handled using hashing (again) and require only a few operations each.

In Ostrum’s original Luthid interpreter every value ever computed was stored in the warehouse, and nothing was ever thrown away. This scheme is very time-efficient (nothing is ever recomputed), but is incredibly wasteful of space. Even the simplest programs soon fill the warehouse with thousands and thousands of tagged items, most of which will never be used again. Ostrum made no attempt to implement a more sophisticated memory-management scheme, but he did leave software ‘hooks’ for one. Provision was made for periodic sweeps of the warehouse, during which items which failed a certain predicate would be discarded. The question was, which predicate?

It is easy to see that in general we cannot predict which stored values will later be needed. We would have to predict the values of the tests in if–then–else expressions. Fortunately, the memory-management scheme does not have to be perfect. If it fails to throw out a few useless items, only a little space is wasted. On the other hand, if it is overeager and discards a few items that are later required, no real harm is done. These values can always be recomputed so that only a little time is wasted. The memory-management scheme can therefore be a heuristic and still be very useful.

For the sake of simplicity, we concentrated on finding one that did not involve analyzing programs. After a few experiments with simple-minded least-recently-used first-out methods, we settled on a heuristic which is reasonably simple, requires no program analysis, and yet performs remarkably well. The heuristic used by the pLucid interpreter is called the retirement plan. It involves discarding those items which experience shows are too old to be of any further use. More precisely, the age of an item in the warehouse is the number of garbage-collecting sweeps that it has survived since it was last used. The age of an item is initially 0. Every time a sweep of the warehouse passes over the item without collecting it, its age is increased by one. The age of an item is stored with the item itself. Different items can of course have different ages.

The interpreter also maintains a “global retirement age” (actually, a retirement age limit). During sweeps of the warehouse, the interpreter discards all those items which have reached the age limit. If this results in an insufficiently large percentage of the items being discarded, the warehouse is enlarged (more storage is demanded from the operating system).

The retirement age limit is quite small and changes dynamically. During computation, every time an item is fetched from the warehouse, a note is made of its age (before this age is reset to 0). If the age in question is greater than or equal to the retirement limit, the limit is increased to the age of the item, plus one. On the other hand, after every sweep, the retirement limit is decreased by one. The retirement age moves up and down and ‘adapts’ to the demand patterns of the interpreter. The provision for increasing it helps avoid retiring items which will be later needed, while the regular decrementation ensures that unused values are soon collected.

The interpreter uses a simple but vital refinement of this scheme: it keeps a separate retirement limit for each distinct program variable. By maintaining a whole vector of retirement limits, it is able to allow for different demand patterns for different variables. In practice, the retirement limits for different variables can be quite different, though they are usually small.

The retirement scheme has proved to be extremely successful, at least in comparison with the strategy of saving everything. Programs often use one tenth the storage or even less, and simple iterative ones (like the square generator given above) run in constant space. The retirement plan is capable, in principle, of discarding values that are later required; but we have never found a ‘sensible’ program in which this ever happens. It seems that the plan takes advantage of the fact that demand patterns are almost chronological (non-anachronistic), even if in general they are not exactly so.
4 Tags in the Interpreter

Eduction is a very simple technique as long as the “tags” involved are sequence indices, i.e., essentially small integers. Complications arise when we try to extend the method to handle two of the most basic features of pLucid, namely nested loops and user-defined functions.

Lucid has a very general facility for nesting of iterative computations. The programmer can write one loop inside another, so that for every step of the outer loop an entire (finite) computation runs to completion (we will omit the details). This is in a sense a very conventional idea, but it means that values of variables can no longer be indexed by a single time parameter; we must use a whole sequence of time parameters, one for each level of nesting. These sequences can be arbitrarily long—as long as the most deeply nested loop in the program.

The second major feature of Lucid, user-defined functions, causes even more difficulty. The value of a variable like

\[ \text{tot}(x) = s \text{ where } s = x \text{ fby } s + \text{next } s; \text{ end} \]

depends on more than just time; it depends on the particular ‘call’ or ‘invocation’ of the function \( \text{tot} \). If we think about \( \text{tot} \) operationally, as a filter, we must imagine that the definition of \( \text{tot} \) is simply a template, and that each separate use of \( \text{tot} \) is a separate copy with its own separate internal storage. As a result, there may, at any given time \( t \), be many different individual values of the variable \( s \).

Fortunately, there is a simple way to assign unique ‘coordinates’ to the calls of user-defined functions in any particular program. The collection of all such calls forms a tree. Each node of this tree is a particular call to a particular function, and the children of the nodes correspond to calls in the body of the definition of the function. Any particular function call can therefore be identified by a path through this tree, to the node in question. This path is a finite sequence of textual occurrences of function calls—what we call the place parameters. The time and place parameters together are enough to determine individual values of variables like \( s \).

The individual place parameters are essentially small integers. The place sequences, however, can be very long—as long as the deepest function call. If there is recursion in the program, the place sequences can be arbitrarily large. Worse still, if nested loops occur in recursively defined functions, the sequence of time parameters can also become larger and larger. A simple-minded implementation of eduction would be forced to compare and manipulate long linked lists at every step, and would be extravagantly inefficient in both space and time.

Fortunately, there is a rather simple technique which solves the problem. The technique is known as hash consing and was used by Ostrum in the original Luthid interpreter. Sequences are represented as lists in the usual way, with tail pointers. When, however, it is necessary to construct a new list, search is first made for a list with the given head and the given pointer to the tail. This search uses a hash table of \((\text{head}, \text{tail pointer})\) pairs and requires only a few steps. The advantage of hash-consing is that comparing two sequences involves a single comparison of two pointers. The list operations are therefore all time efficient, and the extra space required for the hash table is not unreasonable.

5 Input/Output in pLucid

Input/output has always been the Achilles heel of nonprocedural languages. This problem is that I/O is an inherently dynamical activity, whereas functional languages are often thought of as inherently static. Furthermore, output seems difficult to imagine without commands being executed, say to eject a new page, or to move the cursor to the next line on the screen.

In Lucid, however, the programmer is already encouraged to think dynamically. Even when discussing the internal structure of a program it is quite natural to speak of the output of one filter being the input of another. In fact, the output of the program as a whole is a stream, as is its input. The interpreter merely applies the dynamic interpretation of Lucid sequences to the
program’s input and output streams. The program as a whole can be thought of as a filter; the interpreter makes these thoughts a reality.

When the pLucid interpreter is given a program, it begins by computing the value of the program at time 0. When this value is available it ‘outputs’ it, and begins the computation of the time-1 value of the program. This is eventually output, as are the values at times 2, 3, 4, …, and so on.

The demands for output values propagate back through the program and eventually produce demands for input values. The only problems are that the input demands may not be needed at all. The interpreter could use some sort of eductive input scheme, but that would interface poorly with humans (not to mention operating systems). Programmers could become quite disoriented with the questions: “What is the third character you will type in? Thank you! Now what is the second character you will type in? . . . .”

The interpreter avoids such bizarre behavior by generating extra demands to keep the input time index in strict sequential order. For example, if the first demand generated internally is for the time-2 value of the input variable \( i \), the interpreter first asks for the values of \( i \) at times 0 and 1 as well. These values are stored in the warehouse, so that the first three input values will not be demanded (of the user) again. The interpreter unfortunately has to store input values indefinitely because, once discarded, they cannot be recomputed.

The interpreter does not anticipate demands for input by demanding it any time later than the internal demands. In that sense, the interpreter’s input is still lazy. This amount of laziness is very important, because otherwise it would not be possible to write programs which interact with their users.

Of course, there are no pLucid commands for ejecting pages and the like. The output of a program may, however, be a stream of characters, and this stream (which may include control characters) can be sent directly to the device. If the program generates the proper “escape sequences”, the programmer may enjoy pages being ejected, cursors jumping around, bells ringing, and whatever other activity is desired.

Lucid streams are all formally infinite. Lucid programs therefore are all, from the formal point of view, nonterminating. Sometimes, however, it is necessary to write programs (such as sorting programs) which expect finite input and generate finite output. pLucid allows such programs to be written in a natural way, but without tinkering with the infinitary semantics of Lucid. The interpreter supports a special unusual data object called \( \texttt{eod} \) (for end of data). When the interpreter’s output routine is asked to output \( \texttt{eod} \), it simply closes off the input stream and terminates normally. At the other end, when the user closes off the input stream, any further demands for input are sent \( \texttt{eod} \) as the result.

It is not an error to apply a data operation to \( \texttt{eod} \); but the result is (almost) always \( \texttt{eod} \). This means that most continuously operating programs behave sensibly (terminate normally) when their input is terminated. The \( \texttt{eod} \) input propagates through the program and quickly appears at the output. At the same time, there is one primitive (\( \texttt{iseod} \)) which can be used to detect ‘termination’. With \( \texttt{iseod} \), we can write programs which take special account of the end of the input stream. For example, the program

\[
\begin{align*}
(\text{sum as soon as iseod next } x) \ fby \ eod \\
\text{where} \\
\text{sum } = x \ fby \ s + \text{next } x;
\end{align*}
\]

reads in numbers up to end-of-stream, then prints their sum and terminates.

6 The pLucid/UNIX Interface

Operating-system command languages (“job-control languages”) are usually considered to be the complete opposite of functional languages, at least as far as declarative style and semantic elegance
are concerned. If functional programmers live on Mount Olympus, then JCL programmers are
callowing in Hades.

In the case of UNIX and pLucid, however, the two extremes are not as far apart as we might
expect. Both are based on a form of dataflow; in fact, the terms “stream” and “filter”, which we
employed earlier to describe pLucid programs, were deliberately borrowed from UNIX terminology.
The interpreter was carefully designed to take advantage of the shared concepts to allow the two
forms of dataflow (however different) to cooperate with each other.

The interpreter itself acts as a UNIX filter when running a Lucid program. The input to the
Lucid program is taken from the interpreter command’s standard input, and the output of the
program is sent on the command’s standard output. The pLucid input and output conventions
are completely consistent. pLucid programs can therefore be ‘plumbed’ together: the output of
one can be sent as the input of another.

The interpreter is, in a sense, UNIX’s interface to pLucid. Using the interpreter, UNIX pro-
grammers can write some of their filters in pLucid. pLucid also provides an interface in the
opposite direction: a facility that allows pLucid programmers to use executable UNIX filters in
their pLucid program. The interface takes the form of a pseudo-function called \texttt{filter}. The first
argument of a call to \texttt{filter} is a string. This string is interpreted as a UNIX command; the
command is set running as a separate process, and values of the second argument are calculated
one by one and sent to the process’s standard input. The values produced on the standard output
are taken as the value of the filter call as a whole. The discrepancy between UNIX’s data-driven
model and pLucid’s demand-driven model cause problems with \texttt{filter}, as do UNIX limitations on
the numbers of pipes and processes which can be working at any one time. The \texttt{filter} interface
is nevertheless quite useful, and does show what is possible in principle.

Filter calls can be used ‘uncleanly’ to generate side effects, such as writing to files. Dirty
\texttt{filter} calls are not covered by the denotational semantics of pure pLucid. Nevertheless it is
better to do something dirtily than to not be able to do it at all.

7 Experience with the Interpreter

The present pLucid interpreter was completed about five years ago and since then has been used
extensively by the authors, E. A. Ashcroft, other colleagues, and large numbers of students in var-
ious dataflow or functional programming courses at Arizona State University and the University
of Victoria. The interpreter has also been distributed to researchers in industry and other univer-
sities. As a result, the pLucid interpreter is being used to run thousands of small programs (a few
dozen lines) and a number of larger ones (hundreds of lines long) as well. The larger ones include
a toy interactive distributed airline reservation system, a very simple nroff-like text formatter, and
a vi-like full-screen editor [7].

The language pLucid has therefore proved adequate for a large number of varied applications.
The retirement scheme has proved to be a crude but effective means of solving the memory-
management problem. In fact, without the retirement plan, it would be impossible to develop,
let alone run, the larger programs. The compile-time and run-time error facilities (not described
here) have proven to be absolutely indispensable (the original Luthid interpreter had no such
features). The input/output and system interface have also proven extremely useful because they
allow crucial parts of a task to be done in conventional languages, parts of which could not be
done (at least, not done efficiently) in pLucid.

The experience with the interpreter has made us even more confident that real applications
programming can be done in a functional language. What remains to be shown, though, is that
it can be done efficiently.

In execution speed, the pLucid interpreter compares well with interpreters for Prolog or lazy
functional languages. But it is still slow compared to code produced by a Pascal or C compiler.

Part of the performance problem is inherent in the interpretation process; the implementation
spends much of its time analyzing expressions and subexpressions in order to decide which demands
to propagate, and how to combine the results. Furthermore, the retirement plan requires repeated
time-consuming sweeps of the warehouse. The retirement plan also errs on the safe side, and consequently leaves significant amounts of useless data in the warehouse. This not only wastes space but also degrades performance by increasing the amount of time required for each sweep of the warehouse (garbage collection).

Recently Ragiv Bagai [3] at the University of Victoria implemented a Lucid-to-C compiler for a subset of Lucid. The compiler runs under UNIX. Bagai's compiler produces C code which in a sense emulates the steps the interpreter would take while evaluating the original Lucid form of the program. Bagai's compiler also employs a usage-count scheme similar to the one developed at SRI by E. A. Ashcroft for a parallel Lucid architecture [1]. This scheme predicts the number of times an item stored in the warehouse will be needed (or used). Each item in the warehouse has a field attached, which indicates the remaining usage count. When this count reaches 0, the storage for the item can be recycled 'on the fly', without expensive sweeps of memory. Initial results show that, in conjunction with other optimizations, both time and space requirements can be reduced by an order of magnitude.

It should be possible to extend Bagai's compiler to handle most of pLucid. An order of magnitude improvement in performance would allow programs like the screen editor and the text formatter to run fast enough to be useful.

8 Acknowledgements

This research was supported in part by the National Science Foundation under grant DCR 84–15618 and by the NSERC of Canada.

References


9