Experience with an Applicative String Processing Language

James H. Morris, Xerox, Palo Alto Research Center
Eric Schmidt, University of California, Berkeley
Philip Wadler, Carnegie-Mellon University

Abstract: Experience using and implementing the language Poplar is described. The major conclusions are: Applicative programming can be made more natural through the use of built-in iterative operators and post-fix notation. Clever evaluation strategies, such as lazy evaluation, can make applicative programming more computationally efficient. Pattern matching can be performed in an applicative framework. Many problems remain.

Introduction

Will real programmers ever write applicative programs? Applicative programming is a style that prohibits assignment statements or other operations that have the effect of changing the values of variables; it deals with the problem of side-effects decisively by ruling them out altogether. Pure LISP [McCarthy] is probably the best known applicative language. The λ-calculus [Church] and Kleene's systems of recursion equations [Kleene] are languages in the realm of logic that can be construed as applicative languages. Many people have developed applicative languages or advocated their use;e.g. [Strachey], [Landin], [Friedman&Wise], [Milner], [Burge], [Backus]. The properties of applicative languages—easy to define semantics, mathematical elegance—are appealing primarily to *meta-programmers.* A meta-programmer, by analogy with a meta-mathematician, does not make his living by programming, but rather by studying programming. Only a few people suggest very forcefully that the applicative style is good for programming *per se.* This paper attempts to explore the question further.

Poplar is an experimental language for text and list manipulation. It has been used for testing some ideas for extending the powers of interactive text editors. It has several aspects, but the one we shall emphasize here is the use of the applicative style in more realistic situations than are normally considered. We designed Poplar to encourage applicative programming, and tried to use it in that spirit. A recipe for it might read: start with pure LISP, replace atoms with decomposable strings, add SNOBOL pattern matching, build-in implicit iteration over lists, sprinkle with untried ideas, add powerful primitives like sorting, fold into an APLish, post-fix syntax, and bake until half done.

Poplar was designed and implemented in 1978 by the first two authors [Morris&Schmidt] and has recently been enhanced by the third [Wadler] who had designed a similar language. It has received moderate use: there have been a few hundred pages of program written by about twenty

professional programmers and computer scientists. They had a good display-oriented text editor, but no text-oriented language like SNOBOL [Griswold] or any UNIX facility like AWK [Aho] or LEX [Lesk]. It has received most of its use from people with a clerical task that is regular enough to be tedious, but not recurrent enough to justify a big programming effort in a more conventional language. A typical comment has been: "In a couple of hours I was able to learn Poplar and use it to solve a problem that would have taken much longer otherwise." A few people wrote more serious programs: a report generation system for software projects, a family budget maintainer, a correspondence management system for academic journal editors, a purchase order management system. Large portions of some of these projects have been written applicatively.

Basics of Poplar

Values and functions

Strings are primitive values and are written in quotes; e.g. "A string" and "". Concatenation of strings is denoted by juxtaposition:

"aaa" "bbb" = "aaabbb"

As in SNOBOL, a number is simply a string of digits. The quotes can be omitted: "123" = 123. Addition and subtraction can be written as infix operations.

The special primitive value *fail* plays the role normally played by Boolean values in Algol; conditional expressions test their parameters for being fail or not, rather than true or false:

if p **then** x **else** y = **if** p≠fail **then** x **else** y

Non-primitive values are either lists or functions. Lists are written like ["A", "list"] and []. Lists may be subscripted: ["A", "B"]$_2$ = "B". (In reality, subscription is written as L/i rather than L_i.) A negative subscript -i yields the list with its first i elements removed. ["A", "b", "c", "d"]$_{-2}$ = ["c", "d"]. Lists can be concatenated with the infix operator ',,'.

["A", "b"] ,, ["c", "d"] = ["A", "b", "c", "d"]

The familiar Cons(x, y) operation of LISP can be accomplished with the idiom [x] ,, y which places x in a list of length one and concatenates it with y.

Functions are denoted by lambda expressions except that instead of 'λx.' one writes 'x:'. An expression like ([x,y]: x+y) is an abbreviation for (z: z_1+z_2). The application of functions to parameters is written in post-fix notation using the operator '/'.

3+9 / (t: t t t) = 121212

There is a standard assignment statement 'x ← e;' it is used mostly for defining functions at the top level. The precedence of ":" is such that one can conveniently use post-fixed functions as a sort of assignment statement.

L/x: x+x = L/(x: x+x)

and is equivalent to (x ← L; x+x).

Equality Assertions

To make programs readable there is a checked comment facility. Any function definition can be decorated with a set of assertions which constitute a test evaluation of the function. For example, given the function

x: [x,x]/Conc/Reverse

one can add equality assertions to produce

```
x:                  = "foo"
[x,x]/Conc          = "foofoo"
/Reverse            = "oofoof"
```

which says: If the input is "foo" the value of [x,x]/Conc will be "foofoo" and the final value will be "oofoof".

This idea has worked out well: it is much easier to grasp what a program is doing if a well-chosen example is interleaved with it. The fact that the example is machine-checked makes it more credible than a normal comment. In practice, one needs mechanical aids to generate examples because of all the details (e.g., How many spaces are in " "?) which escape the reader, but not the checker.

Post-fix syntax and built-in iteration

Since applicative programming has been employed mostly by meta-programmers rather than programmers many of the syntactic creature comforts, like for-loops, are absent from applicative languages. The applicative style usually requires the use of many recursive function definitions, one for every loop. To remedy this situation Poplar supplies several built-in iterative operators. String concatenation and the arithmetic operations extend to lists of strings. Three iterative functionals are infix operators: LISP's Maplist, APL's reduction operator, and an operator similar to the μ operator of recursive function theory. Like function application these three operators are written with the function second rather than first.

```
[a, b, c]//f  =  [a/f, b/f, c/f]        (Maplist)
[a, b, c]///f =  [[a,b]/f, c]/f         (Reduce)

x%f = if x/f then (x/f)%f else x        (μ-operator)
```

A sequence of numbers can be generated by the notation

```
4 -- 7 = [4, 5, 6, 7]
```

A list of equal length of lists may be transposed.

```
[[a, b, c], [d, e, f]]/Transpose = [[a, d], [b, e], [c, f]]
```

Transpose is important because it allows one to generalize a non-unary function, f, to work on lists via the idiom

```
[List1, List2]/Transpose//f
```

The combination of built-in iterators and post-fix notation was very successful; succinct applicative programs to do complicated things could be written easily without using recursion. Furthermore, writing such programs became a simple, natural process, rather than a challenge to the intellect.

As an example, consider the key-word-in-context problem discussed by [Parnas]: given a list of book titles like

```
Green Sleeves
Time Was Lost
```

generate an alphabetized list, useful for looking up specific key words:

```
<Green> Sleeves
Time Was <Lost>
Green <Sleeves>
<Time> Was Lost
Time <Was> Lost
```

The procedure is as follows:

Break the text up into lines.
Break each line up into words.
For each line:
    Generate a list of pairs, one for each word, consisting of the word, and a reconstruction of the line with brackets around the word.
Merge all these lists into one big one.
Sort the list by the words.
Discard the words.
Concatenate all the lines to form the final text.

Figure 1 shows the Poplar program to do this, and Figure 2 shows the same program decorated with equality assertions. The major steps correspond to the informal steps above. The character '↘' stands for carriage-return. The functions Lines and Words are patterns, to be discussed later, that split text into lines and words, respectively. Append concatenates pairs of lists; Conc concatenates pairs of strings. The phrase "//2" is a shorthand for "//(x:x_2)". Besides using a non-trivial recursion, the function G makes heavy uses of the implicit iteration of concatenation. The subexpression (" " w_{-1}) puts a space at the beginning of each word before they are concatenated by Conc. The subexpression (w_1 " " (w_{-1}/G)) prefixes the current word w_1 to every string in the list that the recursive call of G returns.

Figure 1. A Poplar Program for Key-Word-in-Context

```
KWIC ← (s:
        s/Lines
        //Words
        //(WList: [WList, WList/G]/Transpose)
        ///Append
        /Sort
        //2
        ///Conc);

G ← (w: if w/isnull then [] else
          [ "<" w1 ">" (" " w-1///Conc "↘")]
          „ (w1 " " (w-1/G)));
```

Notice that the informal description of this procedure consists of quite imperative statements while the program itself is entirely applicative! This is the advantage of post-fix syntax. The key to this is that at any point in the program there is only one thing being dealt with, and it plays the role normally played by the state of the machine in an imperative program. Many programs have been written in this style, often interactively. The system allows one to type something like "/F" as a command, and function F is applied to the last thing printed out, and then the result is printed. A transcript is kept, and the user may edit this transcript to produce a program.

Figure 2. The Program Annotated with Equality Assertions

```
KWIC ← (s:    ="Green Sleeves↘Time Was Lost↘"

       s/Lines  = ["Green Sleeves", "Time Was Lost"]

       //Words  = [["Green", "Sleeves"],
                   ["Time", "Was", "Lost"]]

       //(WList: [WList, WList/G]/Transpose)
                ="[[["Green", "<Green> Sleeves↘"],
                   ["Sleeves", "Green <Sleeves>↘"]],
                  [["Time", "<Time> Was Lost↘"],
                   ["Was", "Time <Was> Lost↘"],
                   ["Lost", "Time Was <Lost>↘"]]]

       ///Append  =[["Green", "<Green> Sleeves↘"],
                   ["Sleeves", "Green <Sleeves>↘"],
                   ["Time", "<Time> Was Lost↘"],
                   ["Was", "Time <Was> Lost↘"],
                   ["Lost", "Time Was <Lost>↘"]]

       /Sort    =[["Green", "<Green> Sleeves↘"],
                   ["Lost", "Time Was <Lost>↘"],
                   ["Sleeves", "Green <Sleeves>↘"],
                   ["Time", "<Time> Was Lost↘"],
                   ["Was", "Time <Was> Lost↘"]]

       //2       =["<Green> Sleeves↘",
                   "Time Was <Lost>↘",
                   "Green <Sleeves>↘",
                   "<Time> Was Lost↘",
                   "Time <Was> Lost↘"]

       ///Conc   ="<Green> Sleeves
                   Time Was <Lost>
                   Green <Sleeves>
                   <Time> Was Lost
                   Time <Was> Lost
                   ");


       G ← (w:= ["Time", "Was", "Lost"]
              if w/isnull then [] else
              [ "<" w1 ">" (" " w-1///Conc) "↘"]
              „ (w1 " " (w-1/G = ["<Was> Lost↘",
                                  "Was <Lost>↘"]) )
            = ["<Time> Was Lost↘",
               "Time <Was> Lost↘",
               "Time Was <Lost>↘"]);
```

This program is rather inscrutable, but we believe that translating it to a more conventional notation makes it worse. In Figure 3 the program appears written in an Algol/LISP style of syntax, i.e., changed to a prefix notation with all the maplist and reduce operations explicit. To make the nesting tolerable, we introduced many assignment statements; imagine how the program would look if we eliminated them by back-substituting! Of course the assignment statements give one the opportunity to introduce a mnemonic identifier to describe the intermediate result. Thus the opaqueness of the program is due to the style of expression rather than the syntax of the language.

Figure 3. The KWIC Program Written in Algol/LISP

```
procedure KWIC(s)
    begin
    ListofLines ← Lines(s);
    ChoppedLines ← maplist(ListofLines, Words);
    ListofListsofPairs ← maplist(ChoppedLines,
                λWList.Transpose(WList, G(WList)));
    ListofPairs ← reduce(ListofListsofPairs, Append);
    OrderedList ← Sort(ListofPairs);
    ListofStrings ← maplist(OrderedList, λx.x2);
    return reduce(ListofStrings, Conc)
    end

procedure G(w)
  begin
  if isnull(w) then return [];
  First ← "<" w1 ">"
        reduce(maplist(w-1, λx. " " x), Conc) ")";
  Rest ← maplist(G(w1), λx. w1 " " x);
  return Cons(First, Rest)
  end
```

Problems with the syntax

Poplar's users and potential users had mixed feelings about the syntax. Even aspects we consider successful were not universally appreciated. No one was ever sure what the precedence rules were or should be.

Postfix syntax, even if one likes it, has problems: for functions that were binary, one has an urge to place one of the arguments *after* the function name. This syntactic style has evolved in Smalltalk [Kay]. If one wrote a function-producing function like F ← (x: y: x+y), a call of the function, Y/(X/F), did not look right. It was not obvious how to indent programs; in something like [HugeExpression1, HugeExpression2]/TinyFunction, the function name would get lost.

Instead of providing **if-then-else** expressions (as this paper suggests) we used two binary operations > and | with the following definitions:

x > y = **if** x=fail **then** fail **else** y

x | y = **if** x=fail **then** y **else** x

This allows one to write things like (BigExpression | u) rather than the more cumbersome

t ← BigExpression; **if** t=fail **then** u **else** t

The conventional **if** p **then** x **else** y could almost be achieved by (p > x | y). In retrospect, this syntax caused more confusion than it was worth.

A better set of operators

Although the built-in iterators were successful in general, we now have a better idea of what they should be. The maplist operator had the feature that if a value in the output list was fail it was omitted. This was handy, but occasionally it tended to bury errors one would like to discover. There should be a separate operator to accomplish this, perhaps Split defined as follows:

[l, p]/Split = [sheep, goats]

where sheep is the list of items on list l for which p is true (i.e. not fail) and goats is a list of all the others.

The reduce operator /// was confusing to use when the function was not associative. The following definition would have been more useful:

[x,y,z]///F = [x, [y, [z, []]/F]/F]/F

It processes the list from right to left and includes the empty list in the enumeration. This would allow us easily to solve the bothersome problem of eliminating adjacent repetitions from a list.

[1,1,3,4,1,2,2]
///([x,y]: **if** (~y/isnull **and** $x=y_1$) **then** y **else** [x],,y)
= [1,3,4,1,2]

Of course, this reduce would have not have worked for functions like Plus and Conc which didn't expect to see the null list as an argument; but such functions could be extended to take lists of parameters as a matter of course making their use with reduce unnecessary.

The general iteration operator, %, was not very useful. Perhaps we should have built in the list iterator described by [Burge] which is something like

Lit f g h a x = **if** f(x) **then** a **else** g x (Lit f g h a (hx))

The need was felt for ways other than the sequence operator to generate lists from whole cloth. For example, the following function might be useful:

[a, f]/GenList = **if** a=fail **then** [] **else**
[a] ,, ([a/f, f]/GenList)

An infix functional composition operator, e.g.

fOg = (x: x/f/g)

would have been used frequently

Clever Evaluation is essential

The KWIC program is inefficient by contemporary standards. Every line seems to create a large new structure which the following line consumes. Great improvements in this algorithm's performance can be made by a little cleverness in the evaluation strategy. We recently changed the implementation to use the lazy evaluation strategy described in [Henderson&Morris] so these multi-pass operations are merged. The essence of the technique is that nothing is evaluated until it absolutely must be. Under this regime lists often behave like streams because their tails remain unevaluated until they are needed. In the case of KWIC the first operator that forces evaluation is Sort which demands that it receive a list of lists each of whose first components is a fully evaluated string. This causes the Append operation to be completed, but the second component of each pair remains unevaluated until the final reduction using Conc. Thus, in principle, this program requires only enough space to create a list of all the individual words and does not require space proportional to its output which approximates the square of the input.

Notice that the revised definition of the reduce operator works much better under lazy evaluation. For example, the beginning of the value of L///Append can emerge before L has been completely traversed.

Since lists are never fully evaluated one can even deal with "infinite" lists. The Fibonacci numbers may be described by the recursively defined list Fib.

Fib ← [1,1] ,, (Fib + Fib_{-1})

Suppose one want to find the first Fibonacci number that is divisible by 3. He can say [Fib,div3]/Search where Search can be defined in terms of Split the obvious way. This will not involve computing any more elements than a more

conventional program would. In general, any **while** loop could be written in this way:

s ← a; **while** P(s) **do** s ← F(s)

can be simulated by

[[a, F]/GenList, P]/Search

Our implementation of lazy evaluation has not been a complete success for reasons which we shall discuss latter, but it has allowed us to be hopeful that this style of programming may someday be more practical.

Pattern Matching

There are two aspects to the design of pattern matching: the parsing of strings and the post-processing of successful parses. We devoted most of our effort to the second of these, on the theory that a great deal is known about the first.

In essence, the matching sub-language is the language of regular expressions. A primitive pattern is either a string or the *ellipsis* '...' which matches anything (like SNOBOL's ARB). Larger patterns may be constructed from smaller ones by using four combination rules: if P and Q are patterns, then so are the following

P Q	concatenation
P | Q	alternation
P† = P | P P | P P P etc.	iteration
P? = (P | "")	optional

The Kleene star pattern P* can be written as P†?. Every pattern is enclosed in braces '{}'. Since patterns can be assigned to variables it is possible to create recursive patterns. For example,

E ← {digit† | "(" E "+" E ")"}

A simple parsing algorithm causes problems

Rather than use a general parsing algorithm like [Earley]'s we chose an *ad hoc* matching algorithm of the no-back-up variety. We felt it would be a lot of work to implement a general parsing algorithm that would run as fast as an *ad hoc* one. Furthermore, it was not clear what to do with multiple parses. Some of the advantage of having a formally correct parser would be lost if the programmer had to understand the matching algorithm in order to decide which parse would come out first. Nevertheless, in retrospect, we feel that a better algorithm is called for because even the implementor found he made mistakes in writing patterns. For example, he would occasionally write something like {("a" | "an") " " Noun} even though the manual stated that this would not work because the matcher would not back up to try the "an" alternative after matching the "a" in a string like "an owl".

The troublesome ellipsis

The ellipsis pattern, which was very handy to use in practice, raises some problems we don't know how to solve, even with a fully general parser, because it gives rise to a considerable amount of ambiguity. The pattern {... "x" ...} can match the string "bxbxb" in two different ways. We chose the shortest-match-first approach so that the string would parse into "b", "x" and "bxb". However, in more complex situations things do not work out well no matter what rule one adopts. Consider the following description of text in which spaces and carriage-returns are used to describe the two-level that appears in the KWIC example.

{((... " ")†? ... "↵")†}

There are many possible parses of the string

"AAA BBB↵CCC DDD↵"

and we cannot think of any consistent rule which will

produce the parse one wants. It seems clear that in this context one intends ellipsis to mean "any characters other than space and carriage-return.". SNOBOL has an expression, break(" "), that means precisely that; and now we appreciate it! In practice, this difficulty has been surmountable; we use a two-step process described below: break up the text at all carriage-returns, then break the sub-pieces at spaces.

Applicative post-processing is workable

How can one make pattern matching an applicative operation? Specifically how does the language make the results of parsing available to post-processors without using side effects? For example, the SNOBOL pattern

P = ARB . X ";" ARB . Y ";"

assigns the parts of the string which fall before the semicolons to the variables X and Y as a side effect of the matching process. This is unsatisfactory because a reader who sees only the name P in a matching operation cannot easily discover what variables, if any, will be changed.

The basic idea in Poplar is that a pattern is a function which can be applied to a string; the result can be fail or something derived from the string by a set of pattern composition rules. As the default, the matcher simply re-concatenates the pieces matched so that

"aa;bbc;"/{... ";" ... ";"} = "aa;bbc;"

However, by decorating the pattern appropriately one can arrange for different things to happen: Suffixing a component with # causes whatever it matches to be discarded.

"aa;bbc;"/{... (";"#) ... (";"#)} = "aabbc"

One can replace pieces by suffixing the phrase "> newpiece"

"aa;bbc;"/{... (";" > "X") ... ";"} = "aaXbbc;"

One can make lists out of the pieces by inserting brackets and commas in the pattern

"aa;bbc;"/{ [... ";" , ... ";"] } = ["aa;", "bbc;"]

Conceptually, it is best to think of a two-phase process: first the string is parsed, then one computes the result from the parse tree using the various signals attached to the pattern. Although it can be syntactically confusing to intertwine these two processes, it overcomes the fact that any division of the two phases can lead to them becoming inconsistent.

Experience suggests a slightly different design for post-processing patterns might be better. First, one is always writing # after string constants to indicate that they should be discarded; the default should be the other way around. Second, including names for the interesting sub-pieces within the pattern has great mnemonic value. Once there are more than two or three interesting parts of a pattern one begins to lose track of the order. The design alternative we now favor was the one chosen by [Wadler]: introduce Pascal records into the language and allow the result of a match to be a record. For example, the value of a match using

{X:: ... ";" Y:: ... ";"}

would be a record with components X and Y. This retains the applicative nature of pattern matching while regaining the virtues of SNOBOL's conditional value assignment notation.

A more significant problem is associated with iterated patterns like {P†}. A SNOBOL programmer can not use the equivalent pattern, ARBNO(P), if he wants to do anything with the result of the parse. If he wants to apply the procedure F to each substring P matched he must write an explicit loop that chops off a prefix of the string matching P, applies F, and starts over. This is too bad: there is a nice construct that can describe iterated

structures, but one must resort to traditional programming to actually process them.

The first solution to this problem is to introduce a new operator '‡' that parses things just like † but produces a list of the items matched rather than re-concatenating them. Then the operation F can be applied to each element on the list using Maplist. Thus one says

string/{P‡}//F

A second answer is given by a very general method for processing the outcome of a pattern match: attach a function to a pattern element and apply it to the result of matching that element. One says

{(P \ F)†}

and the result of a successful match would be computed by applying F to each of the sub-strings which matched P and concatenating the results. This method is applicable in more general cases typified by the recursive patterns. Without functional attachment such patterns are not useful if one wants to process the recursive structure. For example, to parse an expression and compute its value one can write.

E ← {digit! | ["("# E , "+"# E ")"#] \ Plus}

which is succinct if nothing else. Functional attachment was used extensively to build powerful patterns which simultaneously matched and transformed their input.

Multi-pass parsing is conceptually easier, but needs help

Experience has shown that the create-a-list-and-process-it method is usually easier to use than the function attachment method. It seems simpler to comprehend because it is less intricate. In general, the APL style of processing aggregates seems just as appealing for parsing as for list processing. Let us now consider the problem of writing the two patterns Lines and Words that appeared in the KWIC example. Lines is relatively easy

Lines ← {(... "␤"#)‡}

Notice that the carriage-returns are discarded. Words is harder because one has to get the piece immediately following the last space and cope with the case in which there are no spaces.

Words ← {(...# Letter†)‡ ...#}

where Letter is a pattern matching any letter.

If lazy evaluation methods were extended to pattern matching this method would compete with a left-to-right parser. Unfortunately, we found that the semantics we choose for pattern matching are not quite right for lazy evaluation. For example, the value of

S/{P‡ "Z"#}

is fail if S does not end with "Z". Thus one cannot begin to process a long file of P's for fear that the file will not end in "Z".

Because "breaking up" text is a very common operation and our pattern-matching language doesn't seem to do it very gracefully, we contemplate adding it as a primitive.

S / breakup{Separators}

is defined as returning two lists. The first is the list of separated objects, and the second is the list of separators. For example:

"12,4,78"/breakup{","} =[["12", "4", "78"], [",", ","]]
"a12c3"/breakup{digit} =[["a","","c", ""], ["1", "2", "3"]]
"abcd"/breakup {digit} = [["abcd"], []]
Lines ← s: (s/breakup {"␤"})$_1$
Words ← s: (s/breakup {letter†})$_2$

Notice that breakup always succeeds so it is amenable to lazy evaluation.

Implementation Notes

An interpreter for Poplar was implemented on the Alto [Thacker], using the language Mesa [Mitchell]. It is organized so that there is no distinction made between expressions and values. What one normally thinks of as a value is simply an expression that the evaluator will not reduce any further. An expression is represented by a Node and may be one of a variety of different types:

- A string, an empty list, or fail
- A list node with pointers to the first element and the rest of the list
- A specific operator with one or two associated operands; e.g., Plus with a pointer to each summand, or Maplist with a pointer to the function and a pointer to the list.
- A λ-expression
- A closure: a pointer to an environment list of variable-value pairs and a pointer to an expression

The evaluator is a simplifier: passed an expression, it returns a new expression, which is a simplified version of the first. After normal evaluation, an expression will be in one of the following three forms:

- A string or fail.
- A closure of a λ-expression.
- A list composed of the above and (recursively) lists.

To convert the evaluator to be lazy in the manner described in [Henderson&Morris] we made two changes:

- Arguments of a function are not evaluated until needed.
- Components of a list structure are not evaluated until needed.

In each of these cases the expression is put in a closure with the current environment. An outcome of this rule is that the final result of evaluation may be a list node whose components are closures (the suspensions of [Friedman&Wise]). "Needed" means that the value is to be printed or treated as the subject of a pattern match.

We did not make the concatenation of strings or pattern matching lazy, but have chosen a "half-lazy" representation of strings. Both arguments of a concatenation are fully evaluated; but, if the resulting string is more than 100 characters long, the result is represented as a node with pointers to the two strings. Thus, in general, a string is represented by a binary tree of such nodes. The terminal nodes point at pieces of files which are paged in as needed. Immediately before printing or pattern matching, this tree is converted to be right-linear;i.e. each left son is a terminal node. This scheme was arrived at after some experimentation and seems to work well most of the time.

Garbage Collection

We implemented a scan/mark garbage collector for both Nodes and strings. Temporary string storage was compacted, and files were closed if garbage collection revealed that no string pointed to them. We set up strict programming conventions to avoid collection-related bugs. We made it our policy that each procedure would register the address of any local variable of type Node; it did not have to register parameters because they were the caller's responsibility. Registered locations were kept in a stack which grew and shrank in parallel with the Mesa run-time stack. When garbage collection was necessary, only those nodes accessible from registered locations were saved. We used the scan/mark algorithm instead of reference counts because it gave us explicit control over the memory, and no programming errors ever caused us to lose memory since we explicitly confirmed its use every time a garbage

collection happened. If a procedure failed to register a value, the subsequent garbage collection would destroy the values about to be used. Bugs of this sort were not too hard to find since the collector gave nodes on the free list a special type, and subsequent access usually checked the node's type.

Lazy evaluation: surprises and problems

As expected, lazy evaluation required a larger constant overhead than normal evaluation. A lot of time is spent saving contexts in the form of closures and re-establishing them. We guessed that this would cause a slowdown by a factor of three in those computations where one must eventually evaluate everything completely. Happily, it appears that the factor was nearer to two.

Another problem is that the saved closures can tie up a lot of space. To avoid this one can scan the expression part of a closure to determine what variables are free in it, and include only these in the environment list for the closure. We don't know whether this would be worth the bother.

There was an unpleasant surprise in the lazy evaluator design. It sometimes requires twice as deep an evaluation stack as the normal evaluator. Consider

```
Factorial ← ([x, f]: if x=0 then f else [x-1, f*x]/Factorial)
```

One's intuition suggests that this is efficient because the recursive call can be replaced by a simple jump, an optimization that most compilers and some interpreters detect. Unfortunately, under lazy evaluation this program is somewhat less efficient. The problem is that the expression f*x at each level remains unevaluated. Thus when the evaluator gets to the call at which x=0 it begins to work on the expression f to produce a number. At this level f is bound to a closure whose expression is f*x and whose environment binds x to 1 and f to a closure whose expression is f*x etc. In other words, to come up with a numerical value for f the evaluator is going to get into a recursion precisely as deep as the one we thought we were avoiding! This second recursion is not in general avoidable because one doesn't know that * is associative and one is also required to overwrite all those closures with the numerical values on the way back. Furthermore, if the evaluator cannot avoid the recursion in the first place we will need twice as much stack as under a normal evaluator. In practice, this problem is not devastating because Poplar encourages a programming style with no recursion in it whatsoever. One should write

```
Factorial2 ← (x: 1--x///Times)
```

which is shorter, clearer and as efficient under lazy evaluation as Factorial is under normal evaluation, even if we defined 1--x by a recursive procedure.

A way of avoiding some of these difficulties has been suggested by [Turner]. His implementation avoids closures and environment lists entirely by translating the expression into combinators. However, some hand simulations indicate that the size of his combinator expressions may grow large in the same situations that generate many closures under our implementation. His implementation avoids checking each value to see if it is evaluated. It also solves the problem of deeper nesting by expanding the functions in-line the first time they are called.

A more fundamental problem is that lazy evaluation is not as powerful a method of improving performance as one might imagine. Consider the following function:

```
AveragePayroll ← (Payroll:
    Payroll/breakup {")"}//Entry/Salaries:
    [Salaries///Plus, Salaries/Length]/Divide)
```

Evaluation of Salaries///Plus does not require the entire

list Salaries to exist at any one time. Nor does evaluation of Salaries/Length. But since both are to be calculated the entire list Salaries will materialize. Evaluating one forces the list into existence, and it cannot be garbage collected because the other still needs it. There is no mechanism to synchronize the evaluation. In general, this problem may occur whenever a list is generated that needs to be traversed by two different functions. Another example is

[List, P]/Split/[PL, NPL]: [PL///Plus, NPL///Plus]

Problems of this type will often be associated with the reduction operator because it reduces a list to a single value, making greater space savings possible. Writing a few special functions to handle reduction might solve some of these problems. For example, consider reduce:

L/([f1, f2, f3]/reduce) = [L///f1, L///f2, L///f3].

Although its use is not completely natural, one could contemplate a compiler generating it.

Reflections

Lists and Strings should be unified

It never occurred to us at first to unify the concepts of strings and lists; we thought of strings as LISP atoms. However, it became clear that this division forced the language into two pieces as in SNOBOL: the pattern sub-language and the general list-processing language. The shortcomings of this became clear when someone wanted to precede a parsing operation by a lexical analysis that produced a list of strings. The pattern language could not be used on the list! This mistake was avoided in LISP70 [Tesler].

We now contemplate an alternate design in which the base data type is character, and a string is just a list all of whose elements are characters. The puzzle is how to generalize the pattern-matching language so that it works on lists. Now we are required to say what it is about the pattern matching language that makes it so nice other than that it is "just like regular expressions". One thing that makes it powerful is that it is basically a *second order* language like [Backus]'s in that expressions in the language tend to denote functions rather than values. For example, "x y" in the conventional language assumes that x and y denote strings and the value is another string; in the pattern matching language x and y are functions and the result is another function.

Poplar should have a powerful compiler

If we are really going to write programs as profligate as the KWIC example, lazy evaluation is not powerful enough to recover all the efficiency that is needed. The approach demonstrated by [Darlington&Burstall] is more promising and is being studied by the third author who claims that for any function written in a lazy programming style, there is an equivalent and equally efficient program that may be written in the normal style. One can imagine a pre-processor that at compile time performs a source-to-source transformation that converts a lazy program to its non-lazy equivalent. This would avoid the problems discussed above. Furthermore, the compile-time analysis could be used to detect type errors that are especially difficult to cope with when things happen in an order the programmer doesn't expect.

Deep problems about applicative programming

A more serious bar to applicative programming is typified by the following problem: Suppose one wishes to process all the elements of a list, some of which may cause

exceptional conditions. One writes

L // (x: **if** OK(x) **then** newval(x)
else (Exceptions ← Cons(x, Exceptions); x))

The problem is that one wants to use a "side channel" to convey some information which is ancillary to the main computation. In general, if a process has multiple output streams which receive data at very different, unpredictable rates it is difficult to retain an applicative approach.

Beyond this technical problem there are basic, long-standing "philosophical" questions with applicative languages which our experience has brought to the surface:

How should interaction with a user be carried out? In our environment it is the norm to write programs that interact with a person through a keyboard, screen, and pointing device. To describe such things applicatively one can describe each program as a function that maps each "input" into its output response, or better an input *stream* into an output stream as [Friedman&Wise] have done. This model doesn't fit very well with making random changes to a two-dimensional display, however.

How does one debug a program with a surprising evaluation order? Our attempts to debug programs submitted to the lazy implementation have been quite entertaining. The only thing in our experience to resemble it was debugging a multi-programming system, but in this case virtually every parameter to a procedure represents a new process. It was difficult to predict when something was going to happen; the best strategy seems to be to print out well-defined intermediate results, clearly labelled.

How does one predict performance? Never mind that lazy evaluation, or any other clever strategy, will make the program perform better than it would have otherwise—ultimately one depends upon his understanding of the machine to design things so that they run reasonably. If the machine is clever it is probably harder to understand, especially if it employs various *ad hoc* heuristics, based upon expectations of what sort of programs people write.

How does one arrange meaningful checkpoints? Even if one's computation has no bugs and is non-interactive the order in which things are done can be relevant. When one's computation takes a long time he would like to save intermediate states that have meaning to the programmer. For example, in a correspondence management system we found it desirable to produce a letter and record the fact that it had been sent as an atomic action. Typically one might request the system to send many letters and expect that one or two requests would cause trouble for reasons ranging from hardware errors, to software errors, to improper requests. Also, one occasionally wanted to interrupt the process to do something else with the machine. Since there is no interdependence between these requests and because the operation takes a non-trivial amount of time one would like all but the troublesome requests to be completed. We attempted to solve this problem through the use of explicit writes on files—a highly non-applicative operation. If one attempted to describe the operation as a whole, surrendering control of what happens to the system, any mishap forces one to start over entirely.

To summarize, the potential practical benefit of an applicative language is that its implementation has much more running room in which to be clever since the order in which operations are performed is constrained only by the data flow. Examples of such cleverness are lazy evaluation, compile-time loop integration, and parallel processing. On the other hand, computing is an activity which goes on in time and space. In situations where one cares about the time and space aspects of an operation as much as the qualitative result, applicative programming is less applicable(!). Furthermore, the personal, interactive mode

of computing tends to increase the frequency of these situations.

Acknowledgements

Alan Perlis has relentlessly encouraged our exploration of this programming style. Dan Swinehart, Robert Kierr, and Marcello Siero have courageously written programs that they depended upon in Poplar. Paul McJones has made many penetrating comments about the langauge and this paper.

References

[McCarthy] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, Comm. ACM 3, 4 (April 1960) 185-195.

[Church] A. Church, *The Calculi of Lambda Conversion*, Annals of Mathematics Studies, No. 6, Princeton University Press, Princeton, N. J., 1941.

[Kleene] S. C. Kleene, *Introduction to Metamathematcs*, D. Van Nostrand, Princeton, N. J. 1950.

[Strachey] Christopher Strachey, Towards a formal semantics. In Formal Language Description Languages for Computer Programming, T.B. Steel, ed., North-Holland, Amsterdam, 1966, 198-220.

[Landin] Landin, P.J. The next 700 programming languages. Comm. ACM 9, 3 (March 1966), 157-164.

[Friedman&Wise] Friedman, D.P. and, Wise, D.S. CONS should not evaluate its arguments. In Automata, Languages and Programming, Michaelson and Milner, eds., Edinburgh University Press, 1976, 257-284.

[Milner] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, A metalanguage for interactive proof in LCF, in Proc. 5th annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Tucson, Arizona, 1978.

[Burge] William. H. Burge, *Recursive Programming techniques*, Addison-Wesley, Reading Mass., 1975.

[Backus] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21, 8 (Aug. 1978), 613-641.

[Morris&Schmidt] J. Morris and E. Schmidt, Poplar Language Manual, Xerox PARC, internal memorandum, 1978.

[Wadler] Philip C. Wadler, Syntax directed data conversion, Xerox PARC, internal memorandum, 1978.

[Griswold] R.E. Griswold , J.F. Poage , and J.P. Polonsky, *The Snobol-4 Programming Language*, Prentice-Hall, 1971.

[Aho] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, Awk - A Pattern Scanning and Processing Language, Bell Laboratories Internal Memorandum, Murray Hill, N. J., 1978.

[Lesk] M. E. Lesk, and E. Schmidt, Lex - A Lexical Analyzer Generator, Bell Laboratories Internal Memorandum, Murray Hill, N. J., 1978.

[Parnas] D. Parnas, On the criteria to be used in decomposing systems into modules, *Comm. ACM 15*,12, (Dec 1972).

[Kay] A. Goldberg and Alan Kay, Smalltalk-72 instruction Manual, Xerox Palo Alto Research Center, Report SSL 76-6, 1976.

[Henderson & Morris] Peter Henderson and James H. Morris, A lazy evaluator. Third Symposium on Principles of Programming Languages, Atlanta, 1976, 95-103.

[Earley] J. Earley, An efficient context-free parsing algorithm, *Comm. ACM 13*,2, (Feb 1970).

[Thacker] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs, Alto: A personal computer, in *Computer Structures* (second edition), Siewiorek, Bell, and Newell (eds.), McGraw-Hill, to appear.

[Mitchell] J. Mitchell, W. Maybury, R. Sweet, Mesa Language Manual, Version 5.0, Xerox Palo Alto Research Center, report CSL-79-3.

[Turner] D. A. Turner A new implementation technique for applicative languages. Software Practice and Experience 9, 1 (1979), 31-49.

[Tesler] L. Tesler, H. Enea, D. Smith, The LISP70 pattern matching system, Proceedings of the International Joint Conference on Artificial Intelligence, Stanford, 1973.

[Darlington&Burstall] J. Darlington and R. Burstall, A transformation system for developing recursive programs, *JACM 24*, 1, (January 1977), pp 44-67.