RECURSIVE PROGRAMMING THROUGH TABLE LOOK-UP*

Daniel P. Friedman

David S. Wise

Mitchell Wand

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 45

RECURSIVE PROGRAMMING THROUGH
TABLE LOOK-UP

DANIEL P. FRIEDMAN
DAVID S. WISE
MITCHELL WAND

MARCH, 1976

*Research reported herein was supported (in part) by the National Science Foundation under grant no. DCR75-06678 and no. MCS75-08145.
To be presented at 1976 ACM Symposium on Symbolic and Algebraic Computation.

Recursive Programming through Table Look-up*

Daniel P. Friedman
David S. Wise
Mitchell Wand

Computer Science Department
Indiana University
Bloomington, Indiana 47401

Abstract - The maintenance of arbitrarily large tables of previously computed values for functions on integer domains becomes practical when those tables are built using constructor functions which suspend evaluation of their arguments. Two styles of programming with such tables are presented. The first results from replacing recursive invocations within standard recursive function definitions with a reference into a table which is predefined to be all the possible results of the function. The second, more sophisticated, style requires that the table be defined strictly through a generation scheme. In either case the table may be available to the user as a data structure exclusive of the function definition with entries still being manifested only when they are actually used.

Keywords and Phrases - suspended evaluation, generation, streams, infinite lists, dynamic owns, LISP.

CR Categories - 4.13, 4.22, 5.7, 4.12

Introduction

The goal of this paper is to demonstrate that standard recursive definitions may easily be transformed to permit the specification of a table of arbitrary size. The table, built with "suicidal suspensions" [Friedman and Wise 1976], only manifests itself for values which have been explicitly used; these values remain easily accessible for later reuse. The values which are computed from an algorithm expressed in "top-down" recursive code are computed through a bottom-up computation sequence which is time-optimal for that algorithm in the sense of Berry [1976]. Such tables may also be specified by generating functions based on an unconventional formulation of traditional recurrence relations requiring the concept of a suspended structure.

The technique of purely recursive expression was popularized by McCarthy [1960] in the fundamentals of the language LISP, referred to here as pure LISP and elsewhere as LISP 1.0. As this language became more popular, different researchers with varying needs added more and more "enhancements" to the language; notable versions, like LISP 1.5, LISP 1.6, MACLISP, and INTERLISP found their way into common use. In this paper we build on recent results which provide the least-fixed-point semantics for pure LISP, demonstrating new programming styles available within pure LISP with the new semantics. These new styles provide a means for accomplishing many of the goals which motivated the expanded versions of LISP, without the addition of extra structures to the basic language.

The new programming style allows function application as the only control structure and lambda-binding of parameters as the only dynamic association between identifier and value. We provide static bindings, which are not explicitly part of pure LISP, in order to simplify the identification of globally defined functions and globally defined constants. These may be subsumed into classic definitions of pure LISP by regarding such bindings as part of the global association list which exists before run-time. These static globals will be declared explicitly with the use of the symbol "≡".

The concept of suspended evaluation is introduced for LISP by Friedman and Wise [1976] and is similar to lazy evaluation [Henderson and Morris 1976] in its theoretical impact. It is easily implemented by changing the semantics of the elementary functions car, cdr, and cons. Instead of placing the final values in the node allocated by cons, each unevaluated argument with the current environment is placed there in a distinguishable structure called a suspension. The value of a call on cons is unchanged. When either of the two projection functions is invoked on that node, the argument is evaluated in the preserved environment. The resulting value takes the place of the suspension in the node and is returned. As a result the LISP evaluator is invoked no more under this interpretation of elementary functions than under McCarthy's classic interpretation, and in

*Research reported herein was supported (in part) by the National Science Foundation under grant no. DCR75-06678 and no. MCS75-08145.

many instances the evaluator is called significantly less. Thus when the interpreter itself is interpreted under these semantics the least-fixed-point semantics for LISP results.

In this paper we study the effects of this interpretation on programming style and treatment of data structures. This treatment does not depend on a different interpreter, but only on the treatment of the structure manipulation functions car, cdr, and cons. The results generalize trivially to records [Hoare 1975] of more than two fields.

The LISP syntax used here is standard with two minor exceptions. Conditional expressions are presented as alternating sequences of predicates and selections with the commenting keywords if, then, elseif, and else interspersed in order to suggest the pairing which in traditional code is expressed by extra parentheses, creating the appearance of bogus function calls. We have shown elsewhere that under the new semantics cond can be just another function, as it appears in this syntax [Friedman and Wise 1976]. The other syntactic change is related to subscripting of arrays. The form <i lis> where i is an integer and lis is a list returns the ith element in lis.

Infinite Lists

Let us begin with a simple LISP function which no programmer would dare invoke under the old interpretation scheme. Successors is globally defined here as a function and naturals is defined as a constant.

```
(successors n) ≡
    (cons (addl n)
          (successors (addl n)) ) ;

naturals ≡ (successors 0)  .
```

Assuming that the call to successors is not dispatched from the top level of LISP (which includes an implicit print function which traverses the result using car and cdr), we encounter no difficulty in the evaluation of naturals. Its value is a reference to a single node whose A-field, cAr field, is a suspension which would evaluate to 1 if ever coerced and whose D-field, cDr field, is a suspension which would evaluate to another node initially filled with similar suspensions, were it coerced. This structure of naturals is, of course, never seen by the user. Naturals is perceived as a structure to be manipulated, and all reasonable manipulations are well-defined.

For instance, one might ask to see the value of <4 naturals>, the fourth element on the list, sometimes phrased (cadddr naturals). As the interpreter finishes three successive cdr's and then a car, four coercions occur, creating three new nodes and filling in four fields permanently. The result is that 4 is returned, and a change to the invisible physical data structure takes place. A later invocation of <4 naturals> creates no new nodes and causes no coercions. We say the actual data structure is invisible because it reacts to manipulation as if car and cdr were probes into a delicately balanced physical system: whenever a field is accessed its final contents appear.

```
(terms n) ≡ (cons (reciprocal (square n))
                  (terms (addl n))) .
```

This sequence should be familiar to analysts: the sum of the elements (terms 1) converges to $\pi^2/6$. If one made the mistake of printing (terms 1) then the output would begin

```
(1.0  0.25  0.1111  0.0625  0.04        etc.
```

until the capacity of the local machine were exhausted. The sequence is infinite so of course this behavior is to be expected. Under the old interpretation of cons nothing would be printed at all because the traversal of a result by print does not begin until that result is completely manifested as a data structure. Because the suspending cons creates the data structure as the top-down traversal proceeds, we have the net effect of top-down evaluation of recursive code.

Landin [1965] defines a stream as, in effect, a cons which evaluates its first argument but suspends its second. If terms were defined using the stream model then the result of printing (terms 1) would be the same finite prefix of an infinite list as indicated above. The difference is apparent when one considers <3 (terms 0)>. Under the stream model the entire prefix of the list (terms 0) must be coerced up to and including the third element; this coercion diverges because the first element diverges. Under the completely suspending cons, <3 (terms 0)> converges to 1/4 or 0.25 because the value of $1/0^2$ is not coerced in obtaining the value which is the third on the list.

Global Tables

With the elementary concept of infinite lists it is possible to define global structures which are conceptually unbounded but which are, in fact, of a finite length determined by past use. The effect is similar to the dynamic own array of ALGOL 60, but without the associated difficulties. Consider the following function for the ith Fibonacci number, which assumes unused own variables are always initialized to zero.

```
integer procedure Fibonacci(n);
  value n;  integer n;
  begin
    own integer array fib[1:n];
    Fibonacci := (if n < 0 then 0 else
                  if n = 0 then 1 else
                  if fib[n] ≠ 0 then fib[n]
                  else (fib[n] :=
                        Fibonacci(n-2) +
                        Fibonacci(n-1) ))
  end.
```

This procedure has the property that no value of Fibonacci is computed more than once. Thus the sequence of states of the array fib is a time-optimal bottom-up computation sequence in the sense of Berry [1976]. This property is characteristic of the programs we consider.

The procedure runs in time O(n) if the operation of array access is presumed to take constant time. As elegant as that behavior appears when compared with $O(2^n)$ behavior of the classic recursion

else (Fibonacci(n-2) + Fibonacci(n-1))

it is not a fair bound when the problems of dynamic own arrays are considered. If such an array grows larger and larger with successive calls to its enclosing block, the system will either be forced to recopy a sequentially stored vector into a larger buffer, or it will be forced to successively link additional storage onto the pre-existing structure on successive calls. In either case access into such a structure of size n should be charged time O(n), and the net run-time of Fibonacci is $O(n^2)$.

Compare the ALGOL-like code above, with its gross assumptions on initialization of variables and its exclusively local array, to the LISP globals developed below. Due to subscripting conventions,

<n fibtable>

is equal to

(fibonacci (subl n)) .

```
fibtable ≡ (fibtail 0);
(fibtail n) ≡ (cons (fibonacci n)
                    (fibtail (addl n)) ) ;

(fibonacci n) ≡ (cond
    if (lessp n 0) then 0
    elseif (lessp n 1) then 1
    else (sum (fibonacci (sub2 n))
              <n fibtable> )   ) .
```

In order to replace the recursive call (fibonacci (sub2 n)) by a table reference we must be sure that the table entries always exist. This can be accomplished by a minor alteration in the bases:

```
(fibonacci n) ≡ (cond
    if (lessp n 0) then 0
    elseif (lessp n 2) then 1
    else (sum <(subl n) fibtable>
              <n fibtable> ) ) .
```

The structure fibtable is accessible throughout the system. Atoms may be lambda-bound to it; a wasteful user might even attempt to see it, abusing LISP for a peek at its prefix. The time to compute <n fibtable> is bounded by $O(n^2)$, taking into account the O(i) time needed to traverse to the i^{th} element on a list.

We believe that the implementation of infinite structures through suspending cons is far sounder than through other programming styles. The technique of dynamic own arrays is far more powerful and correspondingly more complex than the LISP approach. The contents of dynamic own arrays may be changed at any time while the values within a LISP structure are constant with respect to the binding to the structure, hence pre-ordained, if not already extant. Some care is necessary in constructing the array so that recurrence relations which reference other entries in the array do in fact converge, but this is the same as determining that a highly recursive function definition defines a total function. Such references parallel recursive function calls, asking only a natural extension of LISPers' already refined concepts of recursion and cons, whereas the manipulation of dynamic own arrays require some appreciation of implementation details, such as initialization conventions.

Multi-dimensioned Tables

A recursive (numeric) function of several parameters may use a similarly structured table to retain its values. The dimension is determined by the number of arguments. Interestingly, depending upon the recursion pattern, some entries in this table may never be created no matter what arguments are supplied to the associated function. A familiar example is available in the combination (binomial coefficient) function. Defined in ALGOL and taking into account a well known symmetry it appears as

```
integer procedure Choose (m, n);
  value m, n; integer m, n;
  Choose := (if n < 0 then 0 else
             if n = 0 then 1 else
             if n+n > m then Choose(m, m-n)
             else Choose(m-1, n) +
                  Choose(m-1, n-1)   .
```

This recursion is that often associated with Pascal's triangle, which is the table of values associated with Choose. The subscripting convention again requires that <n <m triangle>> is equal to (choose (subl m) (subl n)); a matrix is represented as a list of vectors so that this expression refers to the n^{th} entry in the m^{th} vector.

```
triangle ≡ (rows 0) ;
(rows m) ≡ (cons (tail m 0)
                 (rows (addl m)) ) ;

(tail m n) ≡ (cons (choose m n)
                   (tail m (addl n)) ) ;

(choose m n) ≡ (cond
    if (lessp n 0) then 0
    elseif (zerop n) then 1
    elseif (greaterp (sum n n) m)
           then (choose m (diff m n))
    else (sum <(addl n)<m triangle>>
              <n <m triangle>> ) ) .
```

As before triangle is a globally defined list which may be inspected or traversed by any function which accesses it as a free variable, but the bindings associated with it cannot be permanently altered. It may be treated as an infinite two-dimensional array, but only part of it will ever be actually created by any single program. In fact, if its only use is as a free variable to choose then only the quarter below a "south-by-southeast" diagonal will ever have manifested values (where the "southeast" diagonal is the main diagonal and all entries above it are zero upon access. See Figure 1.)

To see more clearly which values are created, let us consider some examples. One can evaluate (choose m 0) and (choose m m) without causing any more of triangle to appear than the single node (containing suspensions) which was originally allocated to be its value. Assuming that no entries of triangle have yet been coerced and that $0 < n < m/2$ then a call of (choose m n) will cause the

$$n(m-n) + 2(m-1)$$

nodes to be allocated which build an upper-left corner of triangle (Figure 1) with a capacity for

$$n(m-n) + (m-1),$$

values, but only

$$n(n+1) - \frac{n}{2}(3n-1) - 1$$

values are actually entered. Surprisingly the entry in <(addl n)<(addl m) triangle>> is not made as a result of this call even though the value is at hand.

A dramatically different behavior occurs if the programmer accesses <n <m triangle>> directly rather than calling (choose (subl m) (subl n)). In this case the value is entered into the table and then returned. In this way the zero entries above the main diagnonal of triangle may be called into existence as can never happen when the structure is only accessed via the function choose.

Table accesses are a convenient substitute for recursion whenever the function is commonly used and has the integers as its domain. It is particularly convenient when the recursion pattern is irregular so that the necessary recurrent values are not easily predicted and are supplied through additional clever arguments. Such a recursion pattern is analogous to course-of-values induction.

The programmer, having established a table of the values for a function, is free to access the table or to call the function. If the function contains clever code, such as the symmetry handler within choose, then our advice is to stick with the function. However, if the function requires multiple accesses into the table then the expense of accessing the dynamic data structure can drive the running time bound up to polynomials of degree higher than that warranted by the recursion pattern. In the next section we provide a way to avoid this cost.

Generative Programming

If the programmer uses a table to preserve the values of the function and has that table around at the same time as a potentially useful data structure, then he can define the values in terms of the table alone. We claim that the expression of functions as generated tables is naturally related to the kind of programming exhibited above and we shall demonstrate some advantages of that form of expression.

We briefly present generative forms for the functional tables discussed above.

```
fibtable ≡ (fibtail 0 1) ;

(fibtail m n) ≡
        (cons n (fibtail n (sum m n))) .
```

Now the function call (fibonacci n) is replaced by a table reference <(addl n) fibtable> . Since only one table traversal is performed, the time for all traversals collapses to 0(n) and so the total time becomes linear.

```
triangle ≡ (pascal (cons 1 (zeros))) ;

(zeros) ≡ (cons 0 (zeros)) ;

(pascal row) ≡
    (cons row
          (pascal (cons 1
                        (pairsum row) )) );

(pairsum row) ≡
    (cons (sum (car row)
               (car (cdr row)) )
          (pairsum  (cdr row)) ) .
```

As before <n <m triangle>> is synonomous with (choose (subl m) (subl n)) but without the other definition of choose this computation method, including a single subscripted access, is the only way to recover the binomial coefficients.

Expression of functions through generated tables also allows the table to be built locally and maintained through a temporary lambda-binding instead of through a global declaration. In the examples of the earlier section we accessed the table from within the function which was used to determine other entries in it. This very nearly circular definition was possible because the referenced object was declared to be global at the expense of its being static in interpretation (although not in physical structure). When the same table is expressed generatively its generator may be passed as an argument for local binding and interpretation only within a temporary environment. Upon exit from that inner environment the structure is recovered. A function call like <7 (fibtail 0 2)> returns

26, having established a table for a different Fibonacci sequence which is partially manifested and then lost.

A property of the table generators above is that they do not use the function cond in their definitions. That is a symptom of a very regular recurrence which is not always possible.

As an example of table generated using cond within the code we offer the following closed form for the Sieve of Eratosthenes which generates a list of all the prime numbers without a multiplicative operation. (Henderson and Morris [1976] offer a similar example.) The critical function is removemult which removes values of mult + i·inc for all i from the monotonic list of integers lis, where inc, mult, and lis are its parameters.

```
primes ≡ (sieve (successors 1)) ;
(sieve nums) ≡
  (cons (car nums)
        (sieve (removemult
                        (car nums)
                        (double (car nums))
                        (cdr nums) )) ) ;
(removemult inc mult lis) ≡ (cond
    if (lessp (car lis) mult)
     then (cons (car lis)
                (removemult inc
                            mult
                            (cdr lis) ))
    elseif (greaterp (car lis) mult)
     then (removemult inc
                      (sum mult inc)
                      lis)
    else (removemult inc
                     mult
                     (cdr lis) ) ) .
```

The function call (double n) meaning (sum n n) might be replaced by (square n) by a programmer tolerant of multiplication and familiar with number theory. So the primitive recursive function (prime i) which returns the i^{th} prime is merely <i primes> ; the recursive definition of this function is rather complex without a table or with a table reference used only as a replacement for a recursive call as in the previous section.

Conclusions

The use of an arbitraily large table made possible by the suspending cons was shown to be convenient in efficiently expressing recursive numeric procedures. It is often very easy to replace recursive function calls by accesses into a global structure which preserves the results of earlier calls usable for just the cost of finding them. At this level, such tables may be simulated by dynamic own arrays in block structured languages but at some expense in expression and compiler difficulties.

A more sophisticated use of these structures, although one which asks more than classic recursive definitions of functions, is their use as the explicit function definition. In order to meet this use, the arbitrarily large structure must be expressed as a result of a generating function which is in closed form (exclusive of the table). The computation performance will be slightly more efficient in this form but the required expression may be a problem because such definitions are not in common use. The user can achieve improvements in performance with either choice if the functions are commonly used or highly recursive, just by avoiding wasted effort.

Acknowledgement

We thank Cynthia Brown for her careful reading of the manuscript and for her suggestions which improved it.

References

G. Berry. Bottom-up computation of recursive programs. Revue Francaise d'Informatique et de Recherche Operationaelle 10, 3 (March, 1976), 47-82.

W.H. Burge. Recursive Programming Techniques, Addison-Wesley, Reading, MA (1975).

D.P. Friedman and D.S. Wise. Cons should not evaluate its arguments. Third International Colloquium on Automata, Languages and Programming, Edinburgh University Press (July, 1976).

P. Henderson and J.H. Morris, Jr. A lazy evaluator. Third ACM Symposium on Principles of Programming Languages (January, 1976), 95-103.

C.A.R. Hoare. Recursive data structures. International Journal of Computer and Information Sciences 2, (June, 1975), 105-132.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine - I. Comm. ACM 3, 4 (April, 1960), 184-195.

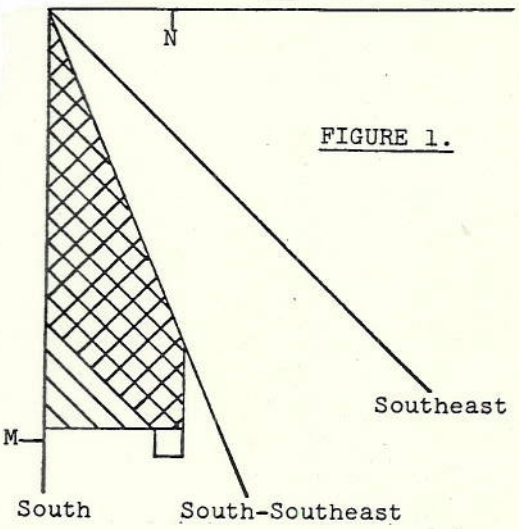

FIGURE 1.

Allocations coerced by (CHOOSE M N) . The hatched area indicates nodes allocated due to coercion of cdr fields. The cross-hatched-area indicates nodes whose car fields are evaluated as well.