FUNCTIONAL COMBINATION

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 27

FUNCTIONAL COMBINATION

DANIEL P. FRIEDMAN

DAVID S. WISE

REVISED DECEMBER, 1976

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

Accepted for publication in Computer Languages.

Functional Combination*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

Abstract - The algebraic functional operation of combination is introduced as a programming tool. It has a practical semantic interpretation in building functions which return several results, especially when such functions are directly recursive. Example functions are given whose invocations build multiple results from single recursions, including a new algorithm for batch-probing binary search trees from an unordered list of keys which returns an ordered list of hits.

Keywords and Phrases - recursive procedures, lists, multiple-valued function, LISP, batch-probing, binary trees.

CR Categories - 4.22, 4.12, 3.74.

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

Introduction

This note advances the concept of functional combination as an important tool for (side-effect free) applicative programming. We have presented the idea elsewhere [2, 3] more completely, imbedded with other recent results in applicative, particularly pure LISP, programming. Moreover, the idea itself is not new [1, 7] although it has not, to our knowledge, been imbedded in a practical programming language.

The problem we address is that of writing a recursive function which returns more than one result. Using the classic _factorial_ recursion as a model, one may observe that recursive functions often contribute something to the answer on each recursion; in _factorial_ it's another factor. We would like to write similar functions which return several such results, each with a contribution from every level of the recurrence.

There are three classic solutions for specifying multiple-valued recursive procedures, none of which meet our demands for style. The first is not even permissible in applicative programming since it requires an assignment to a global (or COMMON) variable or to a parameter called by reference. Additional results may be extracted from recursive functions by communicating them to the calling environment in channels independent of the functions' results, but that is not possible in a regimen, like that of pure LISP, in which all bindings are established specifically through the function linkage (LAMBDA). A second and more obscure alternative involves functions passed as arguments to secondary functions which apply them to other parameters. We would like to avoid such a solution, not because of a ban on second-order functions, which are theoretically sound, but because we would rather provide the programmer with a more stylistically transparent and <u>less</u> powerful tool which he would choose to use freely.

The third classic solution is semantically related to our own. One may write a recursive function whose actual result is always a list or a record of several desired results. The final value of such a function is identical with ours, but the treatment of intermediate results can be messy. In order to make "contributions" to several results, the result's structure must be decomposed and almost immediately reassembled into the same form. Our proposal is semantically equivalent to this alternative, but it does not require the programmer to specify the decomposition and the reconstruction. Imbedded naturally into pure LISP, it interprets a structure in the functional position as an invocation of functional combination with its implicit parameter decomposition and result structuring. The user is able to program according to the answer structure he sees rather than one that he must explicitly and repeatedly describe. Furthermore, a compiling phase can improve the behavior of this code by assigning several registers, one for each constituent of the developing answer, and thus avoiding actual decomposition and reassembly across the recursion.

Functional Combination

We introduce first a bracketing notation for a list function. If a, b, and c are bound to 1, 2, and 3, respectively, then the expression [a b c] evaluates to the list (1 2 3). This is semantically identical to LISP's _list_ function [8].

With the convention that (a*) denotes an infinite list of a's, (a a a ...), the form [a*] evaluates to (1*), the identity vector. The star notation is sufficient for homogeneous infinite suffixes: [a b c*] evaluates to (1 2 3*) = (1 2 3 3 3 ...). More complex infinite data structures are possible [4]. The remainder of the LISP language we use is derived from the S-expressions of McCarthy [8, Chapter 1].

We introduce a list notation for n-tuples which are elements of non-associative Cartesian products of sets W_i. Let $\chi(W_1, W_2, \ldots, W_n)$ be the set of lists of length n corresponding to these n-tuples. $\chi(\chi(W_1,W_2, W_3), \chi(W_4,W_5,W_6))$ is _not_ a set of lists of length 6. Each element is a pair of triples which we shall view as a matrix: two rows of three elements each. When each W_i=W then we can abbreviate $\chi(W_1, W_2, \ldots, W_n)$ as W^n but non-associativity requires that $(W^n)^m = (W^m)^n$ must imply m = n. In all the above definitions n is a non-negative integer or the countable infinity.

We define functional combination in a manner slightly different from the algebraic functor of Cartesian product [7]. The difference appears in the order and the structure of arguments. This definition suits the language LISP since its natural data structure, the list, is accepted by the interpreter as a structure for arguments, as well as by its users who must provide lists as arguments for functional combination.

We extend the definition of a function to be a λ-expression or a list of functions. Anything which functionally evaluates[1] to a list of functions is called a combination. We intend bracketed lists of function-names to be the simplest sort of combination. In McCarthy's [8, Chapter 1] classic LISP interpreter, if the function passed to apply is a list whose car is other than label or lambda, then the result is undefined; what we describe is, therefore, a proper syntactic extension.

Suppose three functions f, g, and h are defined on an r-dimensional domain:

$$f\colon W^r \to W; \quad g\colon W^r \to W; \quad h\colon W^r \to W.$$

Then [f g h]: $(W^3)^r \to W^3$. A legitimate argument structure for [f g h] may be interpreted as a matrix, a list of rows, whose first, second, and third columns provide the arguments for f, g, and h respectively. The structure of the result is a triple of the results of the respective functions. This definition may be generalized [3] to combinations of any length. While parameters are passed from this matrix in "column-major" order, LISP's argument evaluator, evlis, evaluates in "row-major" order. In order to preserve the matrix flavor of argument passing we shall write each argument as a separate line under the combinator, vertically aligned if possible. For example,

[1]Functional evaluation is invoked on the first item in a form being evaluated before it is applied. In some interpreters it is a straightforward evaluation; in others properties (i.e. EXPR, FSUBR, etc.) play a significant role.

evaluation of

```
               ([sum product quotient difference]
                [ 0     3       63         19     ]
                [ 1     3        9         12     ])
yields          ( 1     9        7          7     ).
```

Before proceeding to the next example we introduce the function second as a primitive (like prog2 in LISP) which returns its second argument. We also adopt the convention of giving a multiple-valued function a name hyphenated to suggest the structure of its result.

The next example illustrates the power of functional combination as related to recursive programming. The function ℓt-eq-gt takes a list of numbers and a numeric value as parameters and returns three results corresponding to the three components of the partition of the list by that value: those less than, those equal to, and those greater than it. Since operations like this are common in programming (for example Dijkstra's Dutch National Flag and the key step in the Quicksort [5] Algorithm), it is important that they be expressible in a form analogous to the simple loop available to iterative programmers.

```
          (ℓt-eq-gt ℓ v) ≡ (cond
             if (null ℓ) then [ [] [] [] ]
             elseif (less? (car ℓ) v)
                   then ([cons   second second]
                         [(car ℓ)   NIL    NIL ]
                         (ℓt-eq-gt (cdr ℓ) v))
             elseif (greater? (car ℓ) v)
                   then ([second second cons]
                         [  NIL    NIL  (car ℓ)]
                         (ℓt-eq-gt (cdr ℓ) v))
             else ([second cons    second]
                   [ NIL  (car ℓ)    NIL ]
                   (ℓt-eq-gt (cdr ℓ) v)))
```

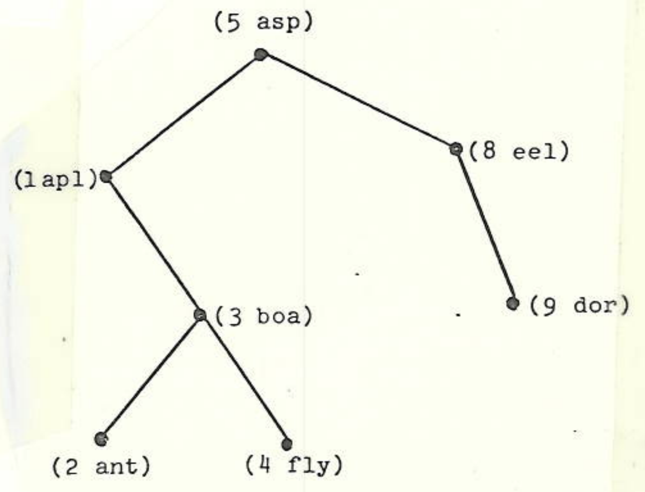
Another application of functional combination involves the invocation of the function being recursively defined with the combinator. Whereas <u>ℓt-eq-gt</u>'s recursions occurred as rows of the argument matrix, these recursions occur as columns. Let ℓ be an unsorted list of perhaps duplicated keys. We present a function <u>quickbatch</u> which probes a binary search [6] <u>tree</u> to extract any information for every key in ℓ, and returns a list of the associations for those keys which had information planted in <u>tree</u>. The list will be returned in ascending order of keys; and the search will be batched [9], so that every subtree is visited at most once.

Define a binary tree to be () or a list of three items: (<u>left</u> <u>information</u> <u>right</u>). <u>Information</u> represents the data stored at the root of the tree whose subtrees are <u>left</u> and <u>right</u>, respectively. In this case <u>information</u> is an association of a <u>key</u> and data. The invocation (key tree) extracts the key from the root of the non-null tree; the definition requires that this key be greater than every key in the <u>left</u> subtree and less than every key in the <u>right</u> subtree.

```
(quickbatch ℓ tree) ≡ (cond
   if (or (null ℓ) (null tree)) then []
   else (apply APPEND ([quickbatch hit quickbatch]
                          (ℓt-eq-gt ℓ (key tree))
                                    tree                 )));

(hit ℓ info) ≡ (cond
   if (null ℓ) then []
   else [info]).
```

The last line of quickbatch deserves some explanation. The result of the use of functional combination is three lists of associations on keys which are to be concatenated. The first and third are derived from recursive calls on the left and right subtrees of the non-null tree. The middle list is empty unless the key found at the root of the tree happened to be mentioned once or more in the target list of the search. Finally, the sorting of the answer list is carried out by an implicit Quicksort [5] at each node in the search tree. The function ℓt-eq-gt partitions at (key tree) the target list carried in an unordered batch to tree. For example, if tree is



then (quickbatch [9 2 3 6 8 7 3] tree) evaluates to ((2 ant)(3 boa)(8 eel)(9 dor)) .

Generalizations

In the examples above we have limited ourselves to combinations of finite length, and to length three in the recursive examples. Moreover, all rows of the argument matrix were of the same length. Now we relax these constraints by allowing the programmer

Generalizations

In the examples above we have limited ourselves to combinations of finite length, and to length three in the recursive examples. Moreover, all rows of the argument matrix were of the same length. Now we relax these constraints by allowing the programmer to provide extra elements in the combination and in the argument rows, with the length of the shortest determining the length of the result. Only the necessary leftmost columns are processed, with superfluous elements being ignored. This convention amounts to a "guillotine rule" on the jagged matrix which results from relaxing the constraints on uniform combination and row sizes.

Infinite rows and infinite combinations are particularly useful under the guillotine rule. Additionally we might usefully define ([f*] [a*] [b*] [c*]) to be [(f a b c)*]. Spreading functions and values across data structures using this syntactic feature is then accomplished with functional combination. The guillotine rule truncates the infinite application. LISP's mapping functions are thereby subsumed by this more general structure. Using our own syntax we provide McCarthy's apply [8, page 13] properly extended to functional combination:

```
(apply fn x a) ≡ (cond
   if (atom fn) then (cond
           if (eq fn NIL) then []              ; COMBINATION IS EMPTY
           elseif (eq fn CAR) then (caar x)
           elseif (eq fn CDR) then (cdar x)
           elseif (eq fn CONS) then (cons (car x)(cadr x))
           elseif (eq fn EQ) then (eq (car x)(cadr x))
           elseif (eq fn ATOM) then (atom (car x))
           else (apply (eval fn a) x a))
   elseif (eq (car fn) LAMBDA) then
         then (eval (caddr fn) (append ([cons*] (cadr fn) x) a))
   elseif (eq (car fn) LABEL)
         then (apply (caddr fn) x (cons (cons (cadr fn)(caddr fn)) a))
   elseif (member NIL x) then []             ; GUILLOTINE RULE
   else (cons (apply (car fn) ([car*] x) a)
              (apply (cdr fn) ([cdr*] x) a))) ; FUNCTIONAL COMBINATION
```

Conclusion

Functional combination is not meant to answer any need of applicative programming beyond style. Like the while statement of sequential programming, which does not add any new semantics to FORTRAN's go to, functional combination has appeal for the applicative programmer who would express himself clearly and concisely without yielding to the clutter of express decomposition, the power of functional arguments, or the pitfalls of side-effects. Specifically, using the two-dimensional coding conventions, it allows the clear and transparent expression of a recurrence which builds up several results, or of a map of a k-ary function across one or more lists through the natural star generalization of combination.

References

[1] W.S. Brainerd and L.H. Landweber. Theory of Computation, Wiley, New York (1974), 18-19.

[2] D.P. Friedman and D.S. Wise. An environment for multiple-valued recursive procedures. Proc. 2nd Programming Symposium, Springer-Verlag, Berlin (to appear).

[3] D.P. Friedman and D.S. Wise. The impact of applicative programming on multiprocessing. Proc. 1976 Intl. Conf. on Parallel Processing (IEEE Cat. No. 76CH1127-OC), 263-272.

[4] D.P. Friedman, D.S. Wise and M. Wand. Recursive programming through table look-up. Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation, 85-89.

[5] C.A.R. Hoare. Quicksort. Comput. J. 5, 1 (1962), 10-15.

[6] D.E. Knuth. Sorting and Searching, Addison-Wesley, Reading, MA (1973).

[7] S. Mac Lane. Categories for the Working Mathematician, Springer-Verlag, New York (1971), 2.

[8] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA (1962).

[9] B. Shneiderman and V. Goodman. Batched searching of sequential and tree structured files. ACM Trans. Database Systems 1, 3 (September, 1976), 268-275.