Uppsala University
Computer Science Department
Datalogilaboratoriet

January 1971

Report No 29

A Proposed Solution to the FUNARG Problem*

by Erik Sandewall

Abstract: This paper is a response to Joel Moses's recent paper, "The Function of FUNCTION in LISP, or ...". We give some examples where the FUNARG feature of LISP 1.5 is relative useful and suggest a computationally efficient implementation of FUNARG. The idea in the proposal is to let a FUNARG-expression create indirect-addressing type bindings on the push-list for variables.

---

* The research reported here was supported in part by the Swedish Research Institute of National Defence (FOA P) under beställn. 010-218:1.

1. Why return closed LAMBDA expressions?

The reader is presumed to have read the paper by Joel Moses, "The Function of FUNCTION in LISP, or Why the FUNARG Problem should be called the Environment Problem". This paper describes a possible solution to this environmental problem. The solution is partial, but (I believe) almost complete. We shall use Moses's concepts and terminology. However, unlike him, we shall presume a thorough knowledge of LISP.

Moses points out that the hard part of the FUNARG problem occurs when a function is returned as the result of some computation. His example is (in LISP notation)

```
define f(x) =  if a = 0 then x else -x

define g(x) = prog( (a) a := 2; return(function(f)))

prog ((a,b,h,)  a := 0;
                h := g(2);
                b := h(3);
                ...   )
```

This example works correctly if function(f) returns a "closed LAMBDA expression" (i.e. in LISP terms, a FUNARG expression). The given example is pure, but it is also artificial. Before we proceed, we shall therefore describe two more practical situations where a competent LISP 1.5 programmer may wish to return a closed LAMBDA expression as the value of a computation.

First example: memoization. Suppose an atom G has been given an EXPR (or function definition) of the form

```
(LAMBDA  (H)  (+++++))
```

where the contents of the lambda-body is left unspecified here. Suppose, further, that the function g is only to be used on atomic arguments; that the lambda-body requires a lot of computer time for its evaluation; and that it has no side-effects. It is then an obvious technique to memoize g, i.e. to save the value of g(h1) on the property-list of h1 under the indicator G the first time g is called with this argument, and to retrieve

it from the property-list each succeeding time. If g has the additional property that g(h) always is distinct from NIL, g can be memoized by changing its definition to

```
(LAMBDA (H) (OR (GET H (QUOTE G))
                (PUTPROP H (+++++) (QUOTE G)) ))
```

where or returns its first non-NIL argument and avoids evaluating the remaining arguments, and where putprop returns its second argument.- Memoization has previously been described in papers by McCarthy (1960?) and Michie (1967).

Consider then the exercise to define a function memoize so that doing

```
(MEMOIZE (QUOTE G))
```

changes the EXPR of G from the old one to the memoized form, or to an equivalent memoized form. Memoize should of course have a general definition, and not merely work for the specific atom G.

One method of doing this is of course to go ahead and write a piece of code which breaks down the original EXPR property and builds up the desired, new one. However, it is preferable to have instead a method whereby the old function definition is embedded in the new one. One reason is that we can then permit the EXPR of G to have an arbitrary structure, and our definition of memoize does not have to know about and account for all possible cases.

Since different functions which are to be memoized may have different lambda variables, we must then re-write the memoized definition on the following, equivalent form:

```
(LAMBDA (Z) (OR (GET Z (QUOTE G))
                (PUTROP Z ((LAMBDA (H) (+++++)) Z) (QUOTE G)) ))
```

This solves the problem, but it immediately introduces a new one: the embedding (LAMBDA (Z) ...) expression is rather hairy, even for this simple example. This may make it inconvenient to program memoize:: a trivial method of writing it is to do

```
(LIST (QUOTE LAMBDA) (QUOTE (Z)) (LIST (QUOTE OR) ...
```

There are ways of overcoming this difficulty, e.g. using subst, but another disadvantage remains: we have to take a lot of free cells for the embedding expression each time we wish to memoize some function. The reason why we cannot immediately use the same embedding expression for all functions (g, etc.) is that (QUOTE G) occurs on a low level, and forces us to make a fresh copy each time. The disadvantage increases as memoization is made more sophisticated, and the size of the embedding expression increases.

The obvious way to solve a problem is to bind a variable m to the name of the function (G, etc.) on a high level in the expression, and to use M instead of (QUOTE G) on lower levels. This can be done in various ways, for example in the hairy way by writing a prog. However, LISP 1.5 offers a conceptually pure way of doing it, namely using FUNARG. The EXPR definition of G is then selected as

```
(FUNARG
    (LAMBDA (Z) (OR (GET Z M)
                    (PUTPROP Z (FN Z) M) ))
   ((M . G) (FN . (LAMBDA (H) (+++++]
```

The structure on the last line is an association-list where M is bound to G and FN to the old function definition of G. Moreover, this FUNARG-expression is generated if memoize is defined as

```
(LAMBDA (M) (PROG (FN)
    (SETQ FN (GET M 'EXPR))
    (PUTPROP M
        [FUNCTION (LAMBDA (Z)
            (OR (GET Z M)
                (PUTPROP Z (FN Z) M]
         'EXPR]
```

It is hard to conceive a simpler way of defining memoize. In this definition, it returns a closed LAMBDA expression.

One might argue against this solution that it is computationally inefficient, and that other methods of binding the variable m should therefore be preferred. However, the notational and conceptual simplicity of

the solution using FUNARG is at the same time a reason to look for more efficient implementations of the FUNARG feature than we have today. (If one can not find efficient methods to handle FUNARG during interpretation, there always remains the possibility of writing a function which "compiles" a FUNARG expression into an equivalent but more efficient expression. However, this "solution" should be a last resort).

Second example: processes. A process is vaguely defined as a function which has some local own variables (in the sense of Algol 60); and which belongs to a class of functions with the same program (function definition, lambda body, or whatever you choose to call it), where different members of the class may have different values in their own variables at a given instant. Typically, each process is evaluated a number of times with a different value each time, and the side-effects which are necessary to achieve this are restricted to the own variables. In the general case, we permit each process to take arguments in each call, but this feature is not always used. - Processes of this kind are especially useful in simulation tasks, and are built into may simulation languages. For a more detailed discussion of processes, see Dahl (1968).

With the background of the memoization example, one method of doing processes in LISP should be obvious: use the FUNARG device. Each process is then a FUNARG expression; its own variables are bound on the a-list which is the second component of the FUNARG-expression, and its program (which only needs to occur in one single copy) is pointed to by being the first component of the FUNARG-expression.

Let us work out the details. Suppose we want to define DP in analogy with DE and DF, so that

```
(DP PRØØ  (OWN1 OWN2 ...) (CALL1 CALL2 ...) (+++++))
```

defines PRØØ to be a process generator. Evaluating an expression

```
(PRØØ  A1 A2 ...)
```

then returns as value a process where the own variables own1, own2, ... are initialized to the values of a1, a2, ... ; where call1, call2, ...

are parameters each time the process is called; and where (+++++) is the form which is to be evaluated and return a value each time the process is called. The value of

```
(PRØØ A1 A2 ...)
```

should then be

```
(FUNARG
    (LAMBDA (CALL1 CALL2 ...) (+++++))
   ((OWN1 . VA1) OWN2 . VA2) ... ))
```

where VA1 is the value of A1, etc. This FUNARG expression can be put away as an EXPR, bound to a variable, or otherwise processed.

The definition of PRØØ which accomplishes this is of course

```
(LAMBDA (OWN1 OWN2 ...)
    (FUNCTION (LAMBDA (CALL1 CALL2 ...) (+++++))) )
```

An adequate definition of DP in LISP 1.5 is therefore to give it a FEXPR of the form

```
(LAMBDA (U V) (PUT (CAR U)
   [LIST (QUOTE LAMBDA)
         (CADR U)
         (LIST (QUOTE FUNCTION) (CONS (QUOTE LAMBDA (CDDR U]
   (QUOTE EXPR) ))
```

It is not possible to make renewed use of FUNCTION in the definition of DP to avoid the list and cons operations, because forms like (CADR U) evaluate into lambda variables (or more precisely, variables for lists of lambda variables).

Let us give two examples of the use of these processes:

(a) If we define

```
(DP SPIT (X) NIL
   (COND (X (PROG1 (CAR X) (SETQ X (CDR X]
```

then the value of

```
(SPIT (QUOTE (A B C D E)))
```

will be a function which on its first call returns A, on its second call B, etc. until E, and which then returns NIL on all successive calls.

(b) If we define

```
(DP ALTERNATE (G H) NIL (PROG (M)
    (SETQ M G) (SETQ G H) (SETQ H M) (RETURN (H)) ))
```

then the value of

```
(ALTERNATE (SPIT (QUOTE (A B C D)))
           (SPIT (QUOTE (1 2 3 4 5))))
```

will be a process which on successive calls returns

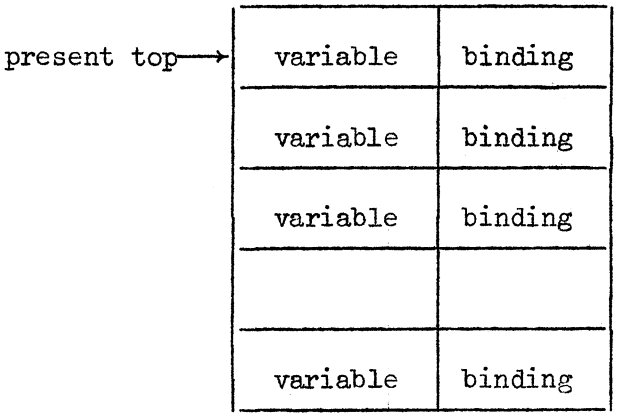
A, 1, B, 2, C, 3, D, 4, NIL, 5, NIL, NIL, ...

Clearly, alternate assumes its two own variables to be bound to processes and uses m for swapping.

In the example for processes, the FUNARG feature which enables us to carry around environments for closed LAMBDA expressions is clearly indispensable. Let us now proceed to the implementation problems.

2. Implementation of FUNCTION in a push-list LISP system

Suppose we have a LISP system where at least logically, variables are bound on a push-down stack ("push-list") of the following form:



In many implementations, the top-most binding of each atom is physically located on this atom's property-list (or in its car), but the logical structure remains the same, at least for the purpose of our discussion.

In such a system, we have problems both when a FUNARG-expression is set up (i.e. when FUNCTION is used), and when the FUNARG-expression is used as a function. Let us consider the problems one at a time, and in that order.

One possibility of handling the problem at "FUNCTION time" (as opposed to "FUNARG time"), is to store away the present push-down stack, e.g. on a secondary memory. This solution has some very obvious disadvantages.

We propose, instead, that FUNCTION shall return a LISP 1.5-style FUNARG-expression, with a conventional association-list containing variable binding, but that this a-list shall only contain bindings of those variables which are given in an optional second argument to FUNCTION ("transfer variables"). Thus the form

```
(FUNCTION FOO (X Y))
```

shall evaluate into an expression

```
(FUNARG FOO  ((X . VX) (Y . VY)))
```

where VX and VY are the values of X and Y in the binding environment. Clearly, if the second argument to FUNCTION is omitted or NIL, FUNCTION can operate like QUOTE (except that the compiler must continue to treat those two differently).

Our proposal then puts some additional burden on the programmer when he wants to use FUNCTION. We believe this is reasonable: in all practical cases of the use of FUNCTION that we have seen, the programmer knows very well what variables in the binding environment he wants to bring to the activation environment, and it is easy for him to give this information to the system. In particular, the two examples in previous section have to be modified as follows:

First example: Add (M) as a second argument to FUNCTION.

Second example: Add (OWN1 OWN2 ...) as a second argument to FUNCTION in the definition of PRØØ. This comes automatically if the generating function DP is changed so that the subexpression in its definition,

```
(LIST (QUOTE FUNCTION
      (CONS (QUOTE LAMBDA) (CDDR U)))
```

goes into

```
(LIST (QUOTE FUNCTION)
      (CONS (QUOTE LAMBDA) (CDDR U))
      (CADR U))
```

Moreover, even with the added nuisance of writing out the transfer variables, it is still much more convenient to use FUNCTION than to use other, dirtier methods for handling these and similar examples.

3. Implementation of FUNARG in a push-list LISP system

The previous section gave us a way to generate good old-fashioned FUNARG-expressions. The remaining problem is: how can we use them in a reasonably efficient way without disturbing the overall efficiency of the system too much. (We do not want those users or those function packages which refrain from the FUNARG convenience to suffer too much, if at all). Let us discuss some possibilities.

First idea: Find the variables of the FUNARG a-list on the push-list for variables. This solution works fine for those cases where the closed LAMBDA-expression only wants to look at the variables that it carries with it, but unfortunately it does not work when the LAMBDA-expression changes (does SETQ on) the same variables. In particular, it does not work for the second example in Section 1 (where FUNARG was used for processes), because the only possible definition of SETQ in this approach is to retain its present definition, where it changes the topmost binding on a push-list. This will have the correct effect within one call of a process (or whatever), but this change of the own variable will not be remembered till the next call of the same process. Therefore, own variables will have their original, initialization value on every call, and the whole point with the process is lost.

Second idea: Let the interpreter maintain both a push-list (for ordinary use) and an a-list (as introduced by FUNARG). When a variable is to be evaluated, check first whether the a-list is empty. If it is, use the push-list as usual. (Thus ordinary users only lose the cost of doing this branch on not NIL every time a variable is to be evaluated). Otherwise, look through the a-list first, and proceed to the push-list only if the a-list contains no binding of the desired variable.

This idea solves the SETQ problem, but it has another, and even worse bug than the previous idea. Suppose we use

```
(FUNARG (LAMBDA (Z) (FOO A Z B)) ((A . VA) (B . VB)) )
```

where FOO is defined as

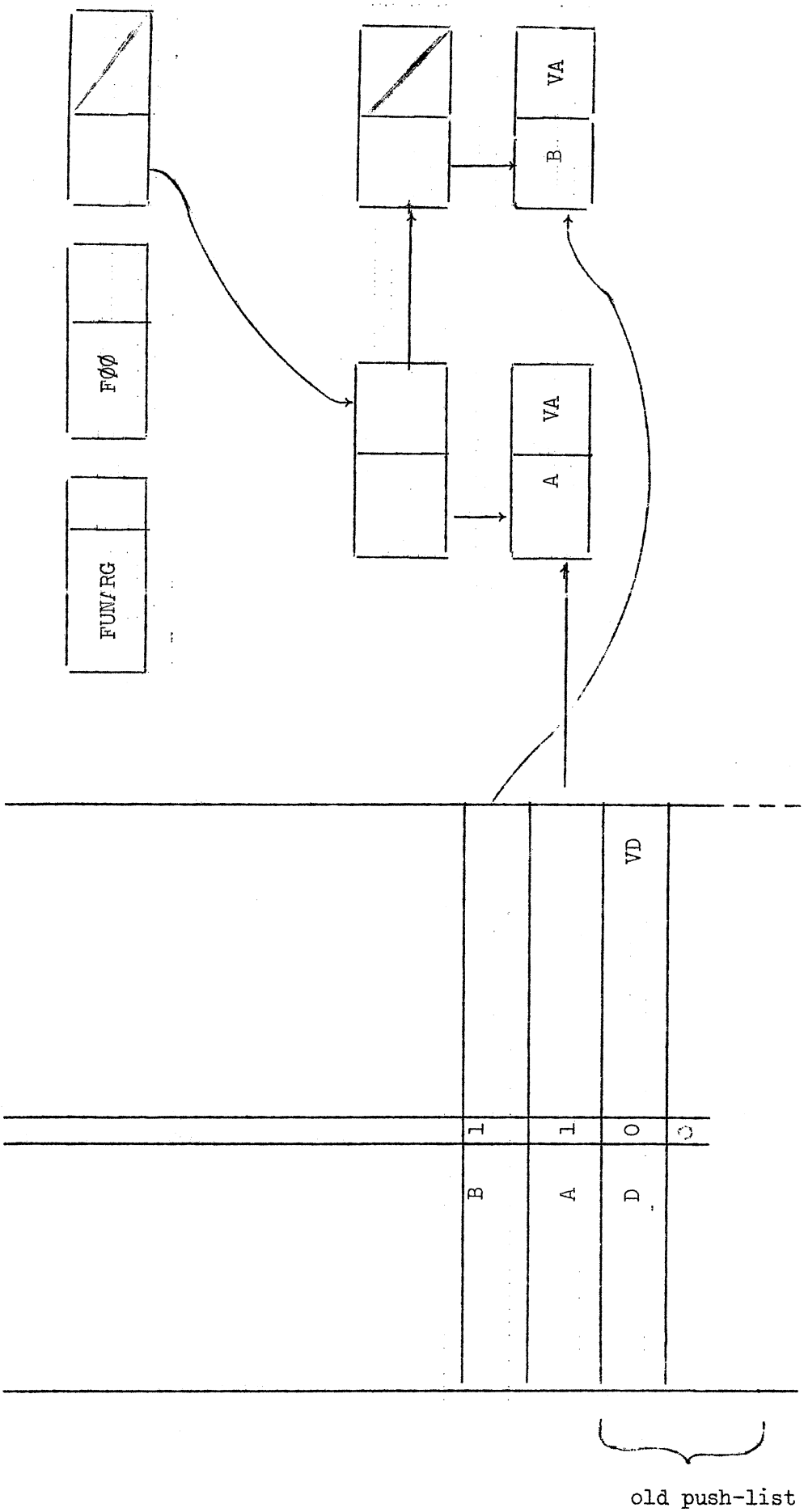
```
(LAMBDA (A B C) (+++++))
```

and the LAMBDA-body uses the value of a, b and c. On the call to foo, these a, b and c will be bound on the ordinary push-list. When (+++++) is evaluated, the a-list bindings of a and b to VA and VB will appear to be "higher up" than the push-list bindings, although in this case they should be "deeper down".

One may try to fix this bug, e.g. by having pointers from the push-list to a-lists which hang around in free memory, or by pushing all variables on the a-list instead of the push-list as soon as the a-list is non-NIL, but such solutions immediately take us back to the inefficiency of LISP 1.5-type implementations. However, they lead us to the

Third idea: Let every level on the push-list consist of two addresses (like before) plus one bit for indirect addressing, which is usually set to zero. Use essentially the first idea of pushing the a-list variables individually, but use the indirect-bit as in the following diagram which describes the bindings for

(FUNARG FØØ ((A . VA) (B . VB)))


FUNARG
FØØ
VA
A
VA
B
VD
1
1
0
0
B
A
D
old push-list

With this solution, we again have one single push-list where all variables are bound, which is fine. There remains some overhead: every time we look up the value of a variable or do a SETQ, we must check the indirect-bit and take slightly different action depending on whether it is on or off. But this is relatively cheap, and it should be an acceptable overhead on most computers. Moreover, in the longer perspective, it is one of the things which could be put into LISP-oriented microprograms when microprogrammed computers become more widely available.

This third idea still is not completely compatible with LISP 1.5. In LISP 1.5 one could write a process which gets hold of its own a-list, and which changes the names of variables (by doing rplaca:s); deletes variable bindings from the a-list (by doing rplacd:s on a-list level): etc. If such a program is to be run on a system which is designed by our third idea, it must be reprogrammed using functions that access the push-list. However, we consider this incompatibility to be of minor importance: whoever uses rplaca or rplacd should be prepared for the consequences. We do believe that the proposals in this paper (including the third idea in this section) do maintain compatibility between LISP 1.5 and push-list LISP for those users who behave decently (by only doing rplaca:s and rplacd:s on what is obviously data), and who are prepared to write out the second argument for FUNCTION. As a simple corollary, it is then a proposed solution to the FUNARG problem.

References

Dahl. Ole Johan, et al. (1968)

Simula 67, Common Base Language
Norwegian Computing Center (Forskningsveien 1 b, Oslo, Norway)
Publication No. S-2

McCarthy, John (1960?)

On Efficient Ways of Evaluating Certain Recursive Functions,
MIT A.I. Project Memo 32

Michie, Donald (1967)

Memo functions: a language feature with rote learning properties.
Research memorandum MIP-R-29.
Edinburgh: Department of Machine Intelligence and Perception

Moses, Joel (1970)

The Function of FUNCTION in LISP or Why the FUNARG problem should be called the Environment Problem.
The ACM Special Interest Group on Symbolic and Algebraic Manipulation,
Bulletin No. 15 (1970)