Short Note

An Approach to the FUNARG Problem Using the MACRO Facility

TAKASHI MURATA* and SABUROU IIDA*

The FUNARG problem is the most troublesome subject for an efficient implementation of Lisp. Usually, this problem is solved by making a function closure of functional arguments. In this paper, we approach this problem using the MACRO facility. Through this approach, the applicability of the MACRO facility to the FUNARG problem has become clear.

1. Introduction

The framework of the programming language Lisp [3] is based on the lambda calculus [2]. For an efficient evaluation of lambda expressions, most Lisp interpreters simulate a substitution operation of the lambda calculus by a variable binding; i.e. to pair variables with their values dynamically. The evaluation by a dynamic binding sometimes causes different results from that of the lambda calculus; a well-known example is the FUNARG problem [4]. In evaluating functionals, the FUNARG problem arises when a conflict of names occurs between free variables of functional arguments and bound variables of functionals. This is due to different environments which determine values of free variables; one is an activation environment of functional arguments and the other is their evaluation environment [6]. On the other hand, when functionals are evaluated after compilation, the FUNARG problem does not arise due to a lexical binding. This semantic gap between interpreter and compiler modes is usually solved by making a function closure of functional arguments.

However, a quite different approach to the FUNARG problem is possible. It is to make use of the MACRO facility. When a MACRO-type function is called, a MACRO expansion is first performed and second an expanded function is evaluated. The essential point of the MACRO facility is that an expanded function is evaluated using the same environment at the time when a function is called. This guarantees the consistency between an activation environment of functional arguments and an evaluation environment of these functions, and suggests the applicability of the MACRO facility to the FUNARG problem. In the following sections, we approach through some examples to the FUNARG problem using the MACRO facility. The method of the approach is to rewrite an ordinary definition of functions to a suitable one to which the MACRO facility is applicable. It is assumed that EXPR- and MACRO-type functions are defined by the notations defun and defmacro, respectively.

2. FUNARG Problem of Non-recursive Functions

First, we investigate non-recursive functions for simplicity. For example, a function func is defined as follows.

```
(defun func (x y fn)
        (list (funcall fn x)(funcall fn y))).        (2-1)
```

Using the definition (2-1), we evaluate the following form.

```
(let ((x '(a b c)))
    (func 'r 's '(lambda (a) (cons a x)))),          (2-2)
```

where a special form let [5] is used. In evaluating the form (2-2), the conflict occurs between a bound variable x in (2-1) and a free variable x of a functional argument in (2-2). In order to keep the consistency between an activation environment and an evaluation environment using the MACRO facility, the rewriting of the definition (2-1) is made as follows. First, following the technique of Common Lisp [5], we replace formal variables with local variables, generated by a function gensym, which are hereafter denoted by a prefix *. Second, we bind actual arguments to these local variables. The function func is then redefined as follows.

```
(defmacro func (x y fn)
  (let ((*x (gensym))(*y (gensym))(*fn (gensym)))
    `(let ((,*x ,x)(,*y ,y)(,*fn ,fn))
```

*Department of Computer and Information Sciences, Toyohashi University of Technology, Toyohashi, 440, Japan

```
        (list (funcall ,*fn ,*x)
              (funcall ,*fn ,*y))))),                    (2-3)
```

where a backquote facility ` is used for readability. Using the redefined (2-3), the form (2-2) after applying the MACRO expansion is represented as follows.

```
let ((x '(a b c)))
  (let ((G1 'r)(G2 's)(G3 '(lambda (a)(cons a x))))
    (list (funcall G3 G1)(funcall G3 G2)))),          (2-4)
```

where G1, G2 and G3 are local variables generated by a function gensym. In evaluating the form (2-4), the evaluation environment of the body of a function func is determined by G1, G2 and G3. However, as these are bound to *r*, *s*, and (lambda (*a*) (cons *a x*)) respectively, the activation environment is closed during the evaluation. This means the consistency between an activation environment and an evaluation environment, and leads to the avoidance of the FUNARG problem.

3. FUNARG Problem of Recursive Functions

In this section, we treat a function mapcar as a representative of self-referential functionals. The definition of mapcar is given as follows.

```
(defun mapcar (x fn)
      (cond ((null x) nil)
            (t (cons (funcall fn (car x))
                     (mapcar (cdr x) fn))))).          (3-1)
```

Using the definition (3-1), we evaluate the following form.

```
  (let ((x '(a b c)))
    (mapcar '(r s) '(lambda (a) (cons a x)))).        (3-2)
```

The FUNARG problem arises similarly to the previous section. However, due to a self-reference of a function mapcar, a new technique is necessary to close an activation environment into the body of mapcar. For the closing, we introduce a local recursion performed by a special form labels [5]. Then, the redefinition of a function mapcar is given as follows.

```
(defmacro mapcar (x fn)
(let ((*mapcar (gensym))(*x (gensym))(*fn (gensym)))
  `(labels ((,*mapcar (,*x ,*fn)
              (cond ((null ,*x) nil)
                    (t (cons (funcall ,*fn (car ,*x))
                             (,*mapcar (cdr ,*x) ,*fn))))))
     (,*mapcar ,x ,fn)))).                            (3-3)
```

Using the redefined (3-3), the form (3-2) after applying the MACRO expansion is represented as follows.

```
(let ((x '(a b c)))
  (labels ((G1 (G2 G3)
             (cond ((null G2) nil)
                   (t (cons (funcall G3 (car G2))
                            (G1 (cdr G2) G3))))))
    (G1 '(r s) '(lambda (a) (cons a x))))).           (3-4)
```

In the above example, the activation environment is closed by a special form labels during the recursive evaluation of the body of a function mapcar.

4. Concluding Remarks

(1) A simple avoidance of the conflict between free variables of functional arguments and bound variables of functions is to make use of the renaming usually made by pre-processors. The overhead arisen due to this renaming seems to be less than that of our method, so our method should be applied to the evaluation of functions after compilation. In spite of this drawback, the characteristic of our method will be sought in the following.

(2) As shown in the sections 2 and 3, our method can close an activation environment into bodies of functions. This closing suggests the applicability of our method to a data flow computing [1], since a data flow computing is principally based on independence on environments; i.e. a binding between variables and values is made dynamically. Although this independence gives rise to the difficulty of the implementation of functionals, our method has shown how to evaluate functionals in the framework of a dynamic binding.

(3) The MACRO facility acts only to the downward direction, and environments disappear after evaluation. In the following example [7],

```
(defun init-count (x) #' (lambda ( ) (setq x (1+x))))
(setq count (init-count 100))
(funcall count)
(funcall count),
```

according to our method, the first funcall sets the value 101 correctly, but the second funcall leads to the incorrect evaluation setting the same value 101. This suggests that only the MACRO facility is insufficient for the upward FUNARG problem.

(4) As the MACRO expansion is used to close actual arguments into places of formal arguments, it is inefficient for such evaluations as (i) functional arguments are determined or redefined dynamically, and (ii) functionals, to which the MACRO expansion is applied, are undefined at the time of compilation.

With respect to (3) and (4) stated above, further investigations will be expected. The examples shown in this paper were tested using mu-Lisp on PC-9801, at that time a special form labels is equivalently replaced using let and funcall.

Acknowledgement

The authors would like to express their sincere thanks to referees for their valuable suggestions.

References

1. AMAMIYA, M. Functional Language and Dynamic Data Flow Machine Architecture (in Japanese), *J. IPS Japan*, **26**, 7, (1985), 765-779.
2. CHURCH, A. The Calculi of Lambda-conversion, Annals of Mathematics Studies, Princeton University Press, New Jersey (1941).
3. MCCARTHY, J. et al. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts (1962).
4. MOSES, J. The Function of FUNCTION in Lisp, *ACM SIGSAM BULLETIN*, (1970), 13-27.
5. STEELE, G. L. et al. Common Lisp, Digital Press, Bedford, Massachusetts (1984).
6. WINSTON, P. H. et al. LISP, 2nd ed., Addison-Wesley, Reading, Massachusetts (1981).
7. YUASA, T. Common Lisp (in Japanese), *J. IPS Japan*, **26**, 7, (1985), 721-731.

(Received July 24, 1986; revised March 12, 1987)